Les langages fonctionnels:Caracteristiques,utilisation et mise en oeuvre

Daniel Le Métayer

Rapports de Recherche

N° 231

LES LANGAGES FONCTIONNELS CARACTÉRISTIQUES, UTILISATION ET MISE EN OEUVRE

Daniel LE METAYER

Août 1983

LES LANGAGES FONCTIONNELS :
CARACTERISTIQUES, UTILISATION ET MISE EN OEUVRE

Daniel LE METAYER (IRISA-LA227 CNRS)
Publication Interne n° 203
162 pages
Juin 1983

RESUME : L'originalité des langages fonctionnels réside dans le fait qu'ils se dégagent de l'influence des machines Von Neumann pour privilégier la facilité d'expression et de preuve des programmes. L'intérêt suscité par ce type de programmation est sans cesse croissant et ce document tente de réaliser une synthèse des travaux effectués ou en cours dans ce domaine. Trois aspects essentiels sont abordés ici :

(i) améliorations apportées par les langages fonctionnels particulièrement en ce qui concerne les raisonnements sur les programmes.

(ii) étude comparative des différents langages fonctionnels.

(iii) mise en oeuvre des langages fonctionnels sous les différents aspects que sont la gestion de la mémoire, le schéma d'exécution, et la structure de machine sous-jacente.

Les quelques problèmes restant ouverts sont évoqués en conclusion.

ABSTRACT: There is an increased interest in the style of programming provided by applicative languages. These languages which have abandonned a lot of Von Neumann features of conventional languages should facilitate the construction and proofs of programs.
This document tries to provide a synthesis of the work which has been performed in this topic. Three main aspects are considered :

(i) description of the essential features of applicative languages.

(ii) comparison between different applicative language.

(iii) implementation of applicative languages. In particular, we insist on aspects such as storage management, run-time organisation, and architectures for applicative languages.

Open problems are discussed in conclusion.

Ce travail a été réalisé dans le cadre de la convention de recherche ADI N° 82/759.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

LES LANGAGES FONCTIONNELS :
CARACTERISTIQUES, UTILISATION ET MISE EN OEUVRE

Daniel LE METAYER (IRISA-LA227 CNRS)
Publication Interne n° 203
162 pages
Juin 1983

RESUME : L'originalité des langages fonctionnels réside dans le fait qu'ils se dégagent de l'influence des machines Von Neumann pour privilégier la facilité d'expression et de preuve des programmes. L'intérêt suscité par ce type de programmation est sans cesse croissant et ce document tente de réaliser une synthèse des travaux effectués ou en cours dans ce domaine. Trois aspects essentiels sont abordés ici :

(i) améliorations apportées par les langages fonctionnels particulièrement en ce qui concerne les raisonnements sur les programmes.

(ii) étude comparative des différents langages fonctionnels.

(iii) mise en oeuvre des langages fonctionnels sous les différents aspects que sont la gestion de la mémoire, le schéma d'exécution, et la structure de machine sous-jacente.

Les quelques problèmes restant ouverts sont évoqués en conclusion.

ABSTRACT: There is an increased interest in the style of programming provided by applicative languages. These languages which have abandonned a lot of Von Neumann features of conventional languages should facilitate the construction and proofs of programs.
This document tries to provide a synthesis of the work which has been performed in this topic. Three main aspects are considered :

(i) description of the essential features of applicative languages.

(ii) comparison between different applicative language.

(iii) implementation of applicative languages. In particular, we insist on aspects such as storage management, run-time organisation, and architectures for applicative languages.

Open problems are discussed in conclusion.

Ce travail a été réalisé dans le cadre de la convention de recherche ADI N° 82/759.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L.A. 227)
UNIVERSITÉ DE RENNES 1 I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

LES LANGAGES FONCTIONNELS: Caractéristiques, construction et mise en oeuvre.

INTRODUCTION

Il apparaît aujourd'hui un nombre de plus en plus élevé de nouveaux langages sans que l'on puisse distinguer le plus souvent l'apport fondamental de chacun. En effet, si l'on observe les langages existants, on constate qu'ils présentent tous de grandes similitudes car ils se caractérisent généralement par une abstraction plus ou moins forte des machines actuelles; par exemple l'affectation d'une variable qui existe dans tous les langages classiques correspond naturellement à une instruction machine de rangement de valeur dans une zone mémoire. C'est en quelque sorte la structure des calculateurs Von Neumann qui a influencé la conception des langages traditionnels. Dans la mesure où il n'existe pas d'alternative à ce type d'architecture, cet état de fait est positif au niveau de la mise en oeuvre car des langages peu éloignés de la machine facilitent une utilisation efficace. Cependant les raisonnements sur les programmes sont complètement sacrifiés dans une telle optique. La meilleure illustration réside dans la complexité et le nombre réduit de preuves qu'il a été possible de réaliser à partir de ces langages. Or cet aspect est primordial dans le but d'une automatisation plus ou moins complète des tâches de conception et de maintenance de programmes.

L'approche de la programmation fonctionnelle est fondamentalement différente puisqu'elle considère en premier lieu la facilité d'exprimer et de prouver les programmes avant d'étudier comment utiliser la technologie en vue d'une éxécution efficace. Il en résulte un type de programmation largement différent dont LISP donne un bon exemple ([MACCARTHY 60]), des possibilités de preuves formelles considérablement supérieures, et des techniques de mise en oeuvre différentes. Nous allons aborder ces différents aspects dans les quatre

parties constituant ce document. La première décrit plus précisément les caractéristiques des langages fonctionnels à travers quelques exemples marquants: LISP, ISWIM et FP. Les améliorations qu'ils apportent par rapport aux langages conventionnels, en particulier en ce qui concerne de l'expression de problèmes et les raisonnements sur les programmes sont analysés dans la seconde partie. La troisième partie constitue une tentative de regroupement et de classification des principaux langages applicatifs existants, tandis que la quatrième est consacrée à la mise en oeuvre des langages fonctionnels sous les différents aspects que sont la gestion de la mémoire, le schéma d'évaluation et la structure de machine sous-jacente. La conclusion enfin, résume quelques problèmes qui restent ouverts et en tire les leçons pour essayer de prévoir l'avenir des langages fonctionnels.

Il a été choisi, afin de ne pas embrouiller inutilement le lecteur par des syntaxes variées, d'exprimer les exemples de cet article en LISP ou en FP, deux des langages fonctionnels les plus référencés et qui sont présentés dans la partie 1.

1ère PARTIE: QU'EST-CE QUE LA PROGRAMMATION FONCTIONNELLE ?

1) Trois exemples de langages fonctionnels.

1.1) LISP

1.1.1) Une seule structure de données : la S-expression.

1.1.2) Les primitives de base.

1.1.3) Les fonctions récursives.

1.1.4) Fonctions d'ordre supérieur.

1.1.5) Conclusion.

1.2) ISWIM

1.2.1) Caractérisation des langages.

1.2.1.1) Quelles sont les manières de composer les objets pour en former de nouveaux ?

1.2.1.2) Quels sont les objets de base ?

1.2.2) Equivalences de base.

1.2.3) Caractéristiques non fonctionnelles du langage.

1.2.4) Conclusion.

1.3) FP

1.3.1) Le langage.

1.3.2) Les primitives.

La nécessité de franchir une nouvelle étape dans la conception des langages a été formulée d'abord par Mac Carthy ([MACCARTHY 60]) et Landin ([LANDIN 66]) avant d'être réaffirmée avec force plus récemment par Backus ([BACKUS 78], [BACKUS 81]). Nous examinons dans un premier temps les critiques qu'ils formulent à l'égard de la programmation conventionnelle et les solutions qu'ils préconisent pour y remédier; les langages qu'ils proposent, respectivement LISP, ISWIM et FP, sont représentatifs de l'ensemble de langages fonctionnels et permettent dans un second temps de dégager les principales caractéristiques de ce type de programmation.

1) TROIS EXEMPLES DE LANGAGES FONCTIONNELS.

Les trois langages décrits ici ont été choisis en fonction du rôle qu'ils ont joué dans le développement de la programmation fonctionnelle: LISP parce qu'il est le précurseur et le plus utilisé des langages fonctionnels, ISWIM à cause de l'importance des idées qu'il a soulevées et FP car il semble l'un des plus prometteurs parmi les récentes propositions.

1.1) LISP

Bien que de nombreux dialectes de LISP existent aujourd'hui c'est la version d'origine de Mac Carthy ([MACCARTHY 60]) qui est prise comme base de discussion ici puisqu'elle contient déjà tous les éléments purement fonctionnels du langage. Nous abordons successivement plusieurs aspects de LISP, à savoir les structures de données qu'il manipule, les primitives disponibles et les manières de construire des programmes à partir des fonctions de base; après quoi quelques points remarquables du langage seront mis en évidence.

1.1.1) Une seule structure de données: la S-expression.

LISP (pour LISt Processor) était conçu à l'origine pour la manipulation d'expressions symboliques, ce qui peut expliquer certaines particularités du langage comme l'uniformité de représentation des données. Toute donnée et tout programme LISP est une S-expression.

La S-expression (expression symbolique ou encore paire pointée) se décrit de la façon suivante:

<S-expression> --> <atome> | (<S-expression>.<S-expression>)

Heureusement, des facilités d'écriture sont permises et les listes peuvent être manipulées d'une façon plus lisible; une liste (A1,...,An) est utilisée pour représenter la S-expression (A1.(A2.(...(An.NIL)...))) où NIL est un atome spécial.

De même les programmes peuvent être écrits sous une forme plus compréhensible, la M-expression où l'application d'une fonction f à des arguments x1,...,xn se note:

f[x1;...;xn].

1.1.2) Les primitives de base.

On peut distinguer les primitives de manipulation de listes (création,accès,...), des primitives booléennes. La première catégorie est constituée de cons, car et cdr. cons permet la construction d'une S-expression à partir de deux composantes tandis que car et cdr produisent respectivement la première et la seconde composante d'une liste différente de NIL. Les primitives booléennes sont atom qui indique si une S-expression est atomique et eq qui, appliqué à deux atomes, teste leur égalité. Notons à cet endroit que CAR, CDR, CONS, ATOM et EQ sont, comme NIL, des atomes spéciaux de LISP: nous adoptons la convention d'utiliser les minuscules dans les M-expressions et les majuscules quand il s'agit d'atomes dans les listes ou les S-expressions.

Exemple 1.

L'évaluation de eq[car[cons[A;B]];A] délivre le résultat "vrai"; cette M-expression peut s'écrire sous forme de liste:

(EQ,(CAR,(CONS,(QUOTE,A),(QUOTE,B))),(QUOTE,A))

et sous forme de S-expression:

```
(EQ.((CAR.((CONS.((QUOTE.(A.NIL)).((QUOTE.(B.NIL)).NIL))).NIL)).
                ((QUOTE.(A.NIL)).NIL)))
```

La primitive QUOTE permet de distinguer les atomes spéciaux des constantes caractères dans les S-expressions. On apprécie mieux sur cet exemple l'intérêt des listes et des M-expressions en ce qui concerne la lisibilité.

1.1.3) Fonctions récursives.

Le moyen de créer une fonction à partir d'une expression est la lambda abstraction empruntée à Church ([CHURCH 41]) et qui se note:

```
\[[x1;...;xn];[E]]
```

Exemple 2.

La lambda expression \[[x];[car[cdr[x]]]] appliquée à (A.(B.C)) rend B; elle se note sous la forme de liste:

```
(LAMBDA,(X),(CAR,(CDR,X)))
```

(on ne détaille plus le passage aux S-expressions qui est automatique et fastidieux).

L'une des particularités de LISP est d'introduire les équations récursives comme modèle de calcul; il est nécessaire pour cela de pouvoir nommer les fonctions (par LABEL) et de contrôler les appels récursifs par l'expression conditionnelle. Ainsi la fonction suivante calcule la longueur d'une liste:

Exemple 3.

```
lg[x]=[eq[x;nil]->0;T->succ[lg[cdr[x]]]]
```

si on dispose de la fonction succ rendant le successeur d'un entier. La traduction sous forme de liste est la suivante:

```
(LABEL,LG,(LAMBDA,(X),(COND,
                       ((EQ,X,NIL),0),
                       (T,(SUCC,(LG,(CDR,X))))))))
```

La valeur de l'expression conditionnelle est obtenue de la manière suivante: les conditions sont évaluées successivement jusqu'obtention d'une valeur "vrai": le résultat est alors le membre droit correspondant. L'effet de LABEL est de modifier l'environnement d'évaluation de la lambda-expression correspondante de telle sorte que le symbole défini (LG ici) y soit disponible, avec comme valeur, la lambda-expression elle-même. Les problèmes posés par l'évaluation sont étudiés dans la quatrième partie.

1.1.4) Fonctions d'ordre supérieur.

Remarquons à cet endroit combien la banalisation des fonctions (comme S-expression) contribue à la puissance du langage; rien n'empêche en effet de les transmettre en argument, ce qui permet de définir sans restriction des formes fonctionnelles, ou fonctions de fonctions.

Exemple 4.

```
mapcar[f;x]=[eq[x;nil]->nil;
            T->cons[f[car[x]],mapcar[f;cdr[x]]]]
```

mapcar prend une fonction et une liste en paramètre et applique cette fonction successivement à chaque élément de la liste.

L'intérêt des fonctions d'ordre supérieur est donc de définir un schéma de programme valable pour un grand nombre d'applications, par exemple dans ce cas ajouter une valeur à chaque élément de la liste ou bien tester si chaque élément est un atome ...

1.1.5) Conclusion

Résumons pour conclure quelques traits caractéristiques de LISP:

- La seule structure de données du langage est la S-expression; les calculs symboliques ([MACCARTHY 60]) ou les preuves de programmes ([BOYER 75]) sont ainsi facilités.

- Les programmes sont des fonctions et ils sont construits par lambda-expression, éventuellement récursivement.

- Bien que le nombre de primitives de base soit faible, il est possible de construire toutes les fonctions calculables en LISP ([BURGE 76]).

- Une conséquence de l'uniformité des structures de LISP est la concision avec laquelle on peut décrire le langage à travers un interpréteur écrit lui-même en LISP ([MACCARTHY 60], [WISE 81]).

1.2) ISWIM

ISWIM (pour If you See What I Mean), décrit par Landin dans [LANDIN 66], est présenté en raison du caractère historique des idées qu'il a soulevées. Nous l'abordons indirectement à travers les principes qu'il met en avant et qui ont été largement repris depuis, notamment dans ML ([GORDON 79]).

1.2.1) Caractérisation des langages.

Les langages sont en apparence tous très proches les uns des autres mais les règles qui régissent l'écriture des programmes varient beaucoup de l'un à l'autre; plus ennuyeux encore, dans un langage donné les conventions ne sont pas toujours les mêmes pour deux types d'objets différents. Un exemple saisissant est celui des identificateurs définis par le programmeur; les contraintes syntaxiques et sémantiques de leur utilisation varient fortement selon le langage et même à l'intérieur d'un langage selon la classe d'objets qu'ils désignent. Il convient donc d'examiner de plus près ce qui caractérise véritablement un langage et quelle est l'importance réelle des choix qui sont effectués. On s'aperçoit en fait que l'on peut séparer la conception des langages en deux étapes: il faut d'abord définir les lois de composition des objets avant de spécialiser le langage par le choix d'un domaine de base précis.

1.2.1.1) Quelles sont les manières de composer les objets pour en former de nouveaux ?

La réponse à cette question permet d'identifier les principales particularités d'un langage. Prenons le cas de la where-expression; par exemple:

x + 1 where x = 4

Pour définir exactement les conditions de son utilisation, il convient d'éclaircir les points suivants: quelles structures linguistiques sont autorisées à figurer dans une where-expression ou à contenir une where-expression: expression conditionnelle, liste d'opérandes, where-expression ...? Quelles sont les contraintes sémantiques sur ces structures? par exemple une fonction est-elle tolérée en partie droite? Quelles sont les règles de

syntaxe: ponctuation, parenthésage...? Remarquons que les réponses à ces trois questions peuvent quasiment être faites indépendamment. Un quatrième point à élucider est bien entendu le sens à attribuer à une telle expression dans tous les cas de figure possibles.

On distingue deux types d'expressions en ISWIM: les a-expressions qui représentent une valeur et les a-définitions qui attribuent un nom à une a-expression. L'utilisation du where est simple: mis à part le fait que partie droite doit être une définition, toutes les combinaisons sont possibles; ceci confère un caractère de simplicité et d'uniformité au langage.

1.2.1.2) Quels sont les objets de base ?

La réponse conditionne l'adaptation du langage à une classe particulière de traitements. Dans cette perspective ISWIM est plutôt une proposition de système d'intérêt général orientable vers un domaine particulier par un choix adéquat de primitives. On peut donc l'assimiler à une famille de langages; cependant Landin ([LANDIN 66]) fait remarquer que ceci revient seulement à formaliser un état de fait; en effet les diverses mise en oeuvres d'un langage reviennent à l'introduction de dialectes tous plus ou moins différents. On éviterait ainsi bien des ennuis en remplaçant le langage par une classe caractérisée par ses propriétés essentielles:

ISWIM est défini à quatre niveaux d'abstraction:

- les ISWIM physiques: implémentés ou définis dans des publications: l'un d'eux est le langage de référence.

- ISWIM logique qui est dégagé de certaines considérations de bas niveau liées à

un type de machine particulier, comme les ensembles de caractères par exemple.

- ISWIM abstrait est le "langage d'arbre" dont ISWIM logique est la linéarisation: il ignore donc les règles grammaticales de regroupement.

- les expressions applicatives (AE) définissent les catégories grammaticales à partir desquelles tous les ISWIM plus étoffés peuvent s'exprimer: c'est donc "le noyau de ISWIM auquel on peut toujours se ramener, le reste n'étant que décoration" ([LANDIN 66]).

Ces expressions sont en fait directement inspirées du lambda-calcul de Church ([CHURCH 41]) et sont définies par un identificateur, par une lambda-expression de la forme (variables liées)(corps de l'expression), ou par une combinaison opérateur-opérande ([LANDIN 64], [BURGE 76]).

1.2.2) Equivalences de base.

La puissance d'expression d'un langage est généralement considérée comme une qualité. Il faut toutefois tempérer ce jugement par l'examen des lois d'équivalence qu'il est possible d'établir à partir des construtions du langage. En effet, si l'accroissement du nombre de possibilités offertes à l'utilisateur pour écrire ses programmes peut paraître une source de souplesse et de puissance (ce qui n'est pas toujours le cas), il s'accompagne le plus souvent d'une complication de la compréhension et des raisonnements sur les programmes. Son intérêt doit donc être clairement défini à la conception du langage, faute de quoi les difficultés introduites au niveau sémantique risquent de compenser largement les facilités pratiques accordées. L'exemple de la procédure est révélateur à ce sujet; en effet, elle peut, contrairement à la fonction, transmettre des résultats par l'intermédiaire de variables globales (effet de

bord); il s'agit effectivement d'une possibilité supplémentaire mais quel est son intérêt ? En realité elle ne fait qu'invalider certaines équivalences, en rendant le sens des procédures dépendant de leur contexte.

Parmi les règles valables pour ISWIM l'une des plus importantes est celle qui permet la transformation des where-expressions:

(L where x = M) ≅

(L en remplaçant dans L les occurrences de x par M).

Un autre type de règles également intéressant est celui des axiomes orientés vers un type d'application particulier; c'est en effet un moyen élégant de spécialiser ISWIM.

1.2.3) Caractéristiques non fonctionnelles du langage.

En fait seulement un sous-ensemble de ISWIM possède les caractéristiques d'un langage fonctionnel. Signalons en particulier une expression appelée "program-point" qui a pour effet de provoquer la terminaison de la where-expression englobante et de lui imposer son propre résultat comme valeur. Il s'agit donc d'une manière de branchement, ce qui est une caractéristique chère à la programmation conventionnelle. Il en résulte entre autres que la loi d'équivalence précédente concernant la where-expression n'est valide que pour le sous-ensemble applicatif de ISWIM.

1.2.4) Conclusion.

Les principales particularités de ISWIM par rapport à LISP sont donc:

- l'absence d'orientation vers un domaine de traitement spécifique (listes dans

le cas de LISP).

- l'utilisation de la where-expression pour faciliter l'écriture (éviter le lourd parenthésage de LISP) mais aussi améliorer la gestion de l'espace mémoire (effet comparable à la structure de bloc).

- l'absence de "trou noir": grâce à la définition à plusieurs niveaux on peut toujours se ramener au noyau de ISWIM; ce n'est pas le cas de LISP et des versions ultérieures ont introduit des caractéristiques contradictoires aux principes de base du langage comme la notion d'adressage ou la modification de zones mémoire ([ALLEN 78]).

- l'existence d'outils conventionnels comme le saut; cependant, si LISP pur ne les admet pas, nombre de ses dialectes ne sont pas non plus purement fonctionnels. Notons au passage la grande similarité du langage utilisé par Burge ([BURGE 76]) pour montrer sur de nombreux exemples comment la programmation fonctionnelle peut être utilisée pour traiter les problèmes courants en informatique (tris, analyseurs syntaxiques,...).

1.3) FP

Backus ([BACKUS 78]) part de l'observation que les grandes similitudes qui existent entre les langages classiques proviennent de leur forte compromission vis-à-vis des architectures Von Neumann. En effet, même si des évolutions ont eu lieu depuis les "langages machine", nombre d'instructions de base du calculateur se retrouvent plus ou moins déguisées au niveau des langages: l'affectation d'une variable est-elle autre chose que le rangement d'une valeur dans une zone mémoire ? Même si ces ressemblances facilitent la traduction du langage dans le code machine, leurs conséquences au niveau de la programmation sont plutôt

néfastes; la complexité des raisonnements sur de tels programmes, par exemple, en donne un bon aperçu.

C'est donc un nouveau style de programmation que Backus entend promouvoir avec son langage FP (pour Functional Programming) qui est volontairenent dégagé de l'influence exercée par une structure de machine particulière.

1.3.1) Le langage.

Un système FP est décrit par les cinq ensembles suivants:

- un ensemble O d'objets.

  Un objet est un atome ou une séquence d'objets notée

  <x1, ..., xn>

  Le symbole ∅ représente la séquence vide.

  Les symboles T, F désignent les valeurs de vérité.

  Le symbole $\underline{[}$ (bottom) désigne l'objet indéfini: il symbolise les cas d'exécution anormale: l'application d'une fonction à des arguments ne faisant pas partie de son domaine de définition, par exemple.

  Toute séquence contenant un $\underline{[}$ au moins est elle-même réduite à $\underline{[}$.

- un ensemble F de fonctions qui transforment un objet en un autre objet. Elles ont la propriété d'être strictes ce qui signifie qu'elles délivrent $\underline{[}$ comme résultat à chaque fois qu'elles sont appliquées à $\underline{[}$. Certaines de ces fonctions sont primitives, d'autres sont construites par l'utilisateur.

- un ensemble C de formes fonctionnelles qui combinent des fonctions et des objets pour produire de nouvelles fonctions.

- une opération qui est l'application d'une fonction à un objet: elle est notée f:x.

- un ensemble D de définitions qui permettent de nommer une nouvelle fonction. Elles ont la forme:

  Def f = fonction construite à partir de primitives et de formes fonctionnelles.

1.3.2) Les primitives.

Voici quelques exemples de primitives fréquemment utilisées en FP; pour une description plus complète consulter la référence [BACKUS 78].

1.3.2.1) Primitives de manipulation de séquences.

- les sélecteurs:

  s:x = si x = <x1,...,xn> avec s <= n alors xs
  sinon ⊥

  où s est un entier.

- distribution à gauche:

  distl:x = si x = <y,∅> alors ∅
  sinon si x = <y,<z1,...,zn>>
  alors <<y,z1>,...,<y,zn>>
  sinon ⊥

- transposition:

  trans:x = si x = <∅,...,∅> alors ∅
  sinon si x = <x1,...,xn>

alors <y1,...,ym>

sinon [

avec xi = <xi1,...,xim> et

yj = <x1j,...,xnj>.

- le rattachenent à droite:

apndr:x = si x = <Ø,y> alors y

sinon si x = <<z1,...zn>,y>

alors <z1,...,zn,y>

sinon [

On ne détaille pas d'autres primitives comme tail, reverse, length et les symétriques des précédentes dont le sens est intuitif.

1.3.2.2) Primitives arithmétiques.

Ce sont les fonctions comme +, -, *, / qui opèrent sur des séquences de deux éléments et délivrent un atome (ou [si l'argument n'est pas convenable).

1.3.2.3) Primitives booléennes.

Ce sont atom, null, dont l'effet est clair et eq qui teste l'égalité de deux éléments.

1.3.3) Les formes fonctionnelles.

Leur importance est capitale puisqu'elles constituent le seul moyen de construire de nouvelles fonctions à partir des primitives.

On suppose par la suite que les symboles f, g, h représentent des fonctions tandis que x et y désignent des objets. Voici les principales formes

fonctionnelles de FP:

- la composition: (fog):x = f:(g:x)

- la construction: [f1,...,fn]:x = <f1:x,...,fn:x>

- la conditionnelle: (p->f;g):x = si p:x = T alors f:x
  sinon
  si p:x = F alors g:x
  sinon ⊥

- la constante: "x":y = si y = ⊥ alors ⊥ sinon x

- l'insertion à droite: /f:x = si x = <x1> alors x1
  sinon si x=<x1,...,xn> avec n >= 2
  alors f:<x1,/f:<x2,...,xn>>
  sinon ⊥

- l'applicaton à tous les éléments:
  ∝ f:x = si x=∅ alors ∅
  sinon si x=<x1,...,xn>
  alors <f:x1,...,f:xn>
  sinon ⊥

- tant que: (while p f):x = si p:x = T
  alors (while p f): (f:x)
  sinon si p:x = F alors x
  sinon ⊥

1.3.4) Quelques programmes FP.

Voici quelques exemples montrant le style de programmation impliqué par ce formalisme.

Le produit scalaire de deux vecteurs s'écrit:

def P = (/+) o (0< *) o trans

Appliquons P à une séquence composée de deux vecteurs; par exemple:

P:<<1,2,3>,<4,5,6>>

trans produit:

<<1,4>,<2,5>,<3,6>>

0< applique la multiplication à chaque sous-séquence rendant:

<4,10,18>

tandisque /+ realise:

+:<4,+:<10,18>>

et son résultat est donc la somme des éléments de la séquence, soit:

32

Notons au passage que l'expression de cet algorithme en FP s'apparente directement à sa formulation dans le langage courant:

(ajouter tous) les (produits des éléments) des (deux vecteurs pris deux à deux)

et qu'il n'est donc nul besoin de dérouler mentalement son exécution pour en comprendre le sens. La même remarque s'applique aux exemples suivants:

Exemple 5.

Le calcul de la longueur d'une séquence peut s'écrire:

Def l = (/+) o (0< "1")

Exemple 6.

```
Def fact = eq0 -> "1"; * o [id, fact o subl].
Def eq0 = eq o [id,"0"].
Def subl = - o [id,"1"].
```

On vérifie facilement que fact:x = x !

1.3.5) L'algèbre de programmes de FP.

FP dispose d'un puissant ensemble de lois sur les fonctions qui est très utile pour l'élaboration de raisonnements sur les programmes. Voici quelques exemples d'axiomes de base:

[f1,...,fn] o g = [f1 o g,...,fn o g]

(p-> f; g) o h = p o h -> f o h; g o h

h o (p -> f; g) = p -> h o f; h o g

p -> (p -> f; g); h = p -> f; h

Il en existe beaucoup d'autres qui sont décrits dans [BACKUS 78], [BACKUS 81], [WILLIAMS 82]. Notons à ce propos l'algèbre développée antérieuremant par Raymond ([RAYMOND 75]) et qui est très voisine de celle de Backus.

1.3.6) Conclusion.

Il apparaît donc que FP est un calcul entièrement basé sur la combinaison de fonctions. Les programmes sont définis de manière statique et dénotationnelle plutôt que par référence à une notion d'exécution liée à un type de machine particulier. De plus, les seules expressions manipulables sont des fonctions et la notion d'objet (argument, variable,...) n'apparaît pas au niveau du langage.

2) ESSAI DE CARACTERISATION DE LA PROGRAMMATION FONCTIONNELLE.

Examinons maintenant les caractéristiques communes de ces langages afin de définir d'une manière plus précise ce que l'on entend par programmation fonctionnelle.

2.1) ABSENCE D'AFFECTATION.

Si LISP et ISWIM possèdent la notion de variable ce n'est pas dans le même sens que les langages conventionnels; il s'agit plutôt d'identificateurs à affectation unique comme les objets définis par une where-expression ou les paramètres formels d'une fonction; en effet la valeur prise par un argument au moment de l'appel ne sera pas modifiée dans le corps de la fonction et lors de deux appels différents on peut considérer que les paramètres formels représentent deux objets différents.

2.2) ABSENCE DE CONTROLE EXPLICITE.

Aucun des langages décrits plus haut n'impose un contrôle précis lors de l'exécution des programmes par des ordres du genre GOTO, EXIT: on dit qu'ils ne sont pas impératifs: l'évaluation de la construction FP ([f1,...,fn]) par exemple n'implique pas un ordre dans l'exécution des fi; la seule dépendance indiquée est celle liée à la logique de l'algorithme: quand celle-ci ne nécessite pas de contrainte d'ordonnancement précise, le langage n'en introduit pas lui-même.

2.3) CALCUL BASE SUR LES FONCTIONS.

Inutile de préciser le rôle primordial joué par les fonctions puisqu'elles constituent le seul moyen de former des programmes en LISP comme en FP; toute la puissance d'expression de ces langages est basée sur les possibilités de les construire et de les manipuler; elles ont donc des prérogatives très importantes: on peut les utiliser comme n'importe quelle autre donnée en LISP (en argument d'une autre fonction par exemple), il est possible de combiner des fonctions pour en obtenir de nouvelles, à l'aide de formes fonctionnelles en FP; la récursivité enfin est largement utilisée puisque l'itération, qui est basée sur la modification de variables, n'existe pas dans les langages fonctionnels.

2.4) CONCLUSION.

Les langages fonctionnels se caractérisent donc globalement par privilégiant une notation plus descriptive des algorithmes par opposition à une vision opérationnelle indiquant étape par étape la démarche à suivre.

2eme PARTIE:

CONSTRUCTION, SIGNIFICATION et MANIPULATION DE PROGRAMMES FONCTIONNELS.

2ème partie: Construction, signification et manipulation de programmes fonctionnels.

1) Construction de programmes fonctionnels.

2) Signification des programmes fonctionnels.

- 2.1) Différents modes de description.
  - 2.1.1) Sémantique dénotationnelle.
  - 2.1.2) Sémantique opérationnelle.
  - 2.1.3) Sémantique axiomatique.
- 2.2) Problèmes sémantiques.
  - 2.2.1) Liaison statique ou dynamique.
  - 2.2.2) Fonctions récursives.
  - 2.2.3) Fonctions strictes ou non strictes.
  - 2.2.4) Les entrées-sorties.

3) Preuves et transformations de langages fonctionnels.

- 3.1) Methodes de preuves inductives.
  - 3.1.1) Induction de Scott (ou du point fixe).
  - 3.1.2) Induction structurelle.
- 3.2) Méthodes sans induction.
  - 3.2.1) L'induction récursive.
  - 3.2.2) Le pliage-dépliage.
- 3.3) Mise en oeuvre de ces méthodes.
  - 3.3.1) Méthodes automatiques.

3.3.2) La méthode semi-automatique de Burstall et Darlington.

3.3.3) Les méta-langages de preuves.

Le meilleur argument en faveur des langages fonctionnels se situe d'après Turner ([TURNER 81a]) dans le gain en temps et en efforts nécessaires pour produire un programme correct. Cette affirmation, qui peut paraître peu fondée voire présomptueuse, a été généralement vérifiée par les utilisateurs de langages applicatifs ([BURGE 76], [BURSTALL 80], [MORRIS 80]) et elle peut en fait s'expliquer plus précisément.

1) CONSTRUCTION DE PROGRAMMES FONCTIONNELS

L'exemple du produit scalaire emprunté à Backus ([BACKUS 78]), établit une comparaison saisissante entre l'expression du problème en FP et dans un langage traditionnel du type d' Algol 60.

Exemple 1.

programme FP: Def IP = (/+) o (∝ *) o trans

programme Algol 60: C:= 0

for I:= 0 step 1 until N do

C:= C + A[I] x B[I]

Quelques remarques s'imposent:

1) Le programme Algol manipule à tout moment un état interne complexe par des règles qui ne le sont pas moins. L'effet d'une instruction dépend entièrement de l'état intermédiaire au moment de son exécution et celui-ci est modifié sans restrictions à chaque étape. Pour prévoir le résultat d'un programme, il faut l'exécuter mentalement depuis le debut pour déterminer l'evolution de l'état après chaque instruction. En FP par contre, les règles se résument à l'application d'une fonction à un argument et d'un combinateur à des fonctions. Les définitions sont parfaitement statiques et une gymnastique de l'esprit n'est pas nécessaire pour en saisir le sens. L'exemple 1 illustre bien l'effet néfaste des variables sur la qualité de la programmation: en Algol il faut identifier le rôle exact et les valeurs prises simultanément par quatre variables avant de tenter de prédire le résultat de la formule. Or

il va de soi que l'initialisation ou le traitement erroné d'une variable a une conséquence directe sur le résultat global. Ce genre d'erreurs est largement favorisé par la programmation traditionnelle et l'usage intensif de variables qu'elle préconise; ici en particulier n'était-il pas tentant de donner pour valeur à N la longueur du vecteur ou d'initialiser I à 1 ...? Remarquons que dans cet exemple, seules A et B représentent les données du problème; les variables I et N qui servent au contrôle ne sont pas impliquées par le logique de l'algorithme du produit scalaire: le séquencement introduit par I est parfaitement artificiel (rien n'empêche de faire deux multiplications simultanément ou dans un autre ordre) tandis que l'introduction de N représente la restriction arbitraire à une taille précise de vecteurs (à moins d'un mécanisme suplémentaire de passage de paramètres) alors que le produit scalaire est défini de la même façon pour tous les vecteurs. Notons également qu'un tableau borné différemment (par exemple [1..n+1]) ne sera pas accepté par ce programme. Les langages traditionnels introduisent une quantité de détails opératoires complètement étrangers à la logique du problème, qui ne font que restreindre et obscurcir la solution et donc favoriser les erreurs.

2) Le programme FP est structuré et hiérarchique: des entités complexes sont construites à partir d'objets de base par un petit nombre de combinateurs; chaque fonction opère sur des structures complètes en entrée comme en sortie. Au contraire les programmes Algol travaillent sur une partie de l'état global et on ne distingue pas de hiérarchie dans le programme de l'exemple 1; la séparation conceptuelle entre la collection de multiplications et l'addition globale est remplacée par une imbrication des deux calculs; on peut dire à la décharge d'Algol que ceci se fait au profit d'un gain d'efficacité puisqu'on

ne parcourt qu'une fois le vecteur et qu'on pourrait réécrire le programme (moins naturellement toutefois) en séparant les étapes.

Le propos n'est cependant pas de prétendre que tout algorithme s'exprime plus facilement et rapidement en FP qu'autrement; il va de soi que l'exemple cité a été sélectionné avec discernement pour montrer combien un choix judicieux de combinateurs (ici /, et ⍀ permet de clarifier l'expression des problèmes: toute la difficulté consiste à faire le bon choix. ([BACKUS 78], [IVERSON 62], [IVERSON 79], [CHIARINI 80]).

2) SIGNIFICATION DES PROGRAMMES FONCTIONNELS.

Les qualités de clarté et de simplicité attribuées aux langages fonctionnels de façon peu rigoureuse dans le chapitre précédent s'expriment plus formellement lorsque l'on considère leur sémantique. Nous examinons d'abord la manière dont les langages fonctionnels sont décrits à l'aide des outils classiques de définition formelle de la sémantique; après quoi sont détaillés les principaux choix à effectuer lors de la définition formelle d'un langage fonctionnel.

2.1) DIFFERENTS MODES DE DESCRIPTION

2.1.1) Sémantique dénotationnelle.

Il n'est pas surprenant que des langages basés sur des notions descriptives et fonctionnelles se décrivent facilement par une sémantique dénotationnelle puisque cette méthode donne un sens aux programmes en leur faisant correspondre une fonction. Pour définir une telle sémantique il faut spécifier les domaines syntaxiques du langage, les domaines sémantiques choisis et les fonctions assurant la correspondance entre les premiers et les seconds.

- Dans le cas fonctionnel les domaines sémantiques fondamentaux sont: les valeurs de base, les fonctions et les environnements qui établissent la correspondance entre les identificateurs et les valeurs des deux autres domaines.

- Dans le cas impératif traditionnel il faut ajouter le domaine des "emplacements", celui des états qui font correspondre une valeur à tout emplacement, celui des commandes et ceux des continuations qui sont nécessaires pour exprimer les ruptures de séquence éventuelles (GOTO par exemple). Inutile de dire que le volume et la complexité de la définition sémantique prennent des proportions telles qu'on peut se demander à quoi elle peut servir ([STOY 81]): elle n'éclaire guère le programmeur sur le sens du langage et l'implémenteur n'y trouve aucune notion opératoire. En effet, des domaines de base, comme la "continuation", ne correspondent pas directement à des éléments du langage familiers au programmeur. En fait ceci constitue plutôt une utilisation contre-nature de la sémantique denotationnelle.

D'après Ashcroft ([ASHCROFT 82]), une démarche plus saine serait de faire jouer à la sémantique dénotationnelle un rôle actif au niveau de la conception de langages (comme dans le cas de LUCID et des langages fonctionnels) plutôt que de s'échiner à lui faire accepter et donc justifier toutes leurs excentricités.

2.1.2) Sémantique opérationnelle.

Spécifier opérationnellement la sémantique d'un langage revient à définir une machine abstraite et à décrire le comportement des programmes sur cette machine; le moyen utilisé dans ce cas ne consiste donc pas à décrire le résultat attendu mais la façon de le calculer. Certains langages fonctionnels ont été définis de cette manière notamment LISP ([MACCARTHY 60]) et ISWIM ([LANDIN 64]); le résultat obtenu est concis et suffisamment clair pour être compris par le programmeur et utilisé avec profit lors de la mise en oeuvre. Il faut cependant comparer ce qui est comparable: la compromission vis-à-vis d'une

machine particulière (même abstraite) n'est généralement pas sans influence sur le sens même donné aux programmes et la sémantique opérationnelle la plus naturelle n'est pas forcément cohérente avec la description dénotationnelle qu'on peut attendre du langage. C'est ainsi que l'on s'est aperçu que la définition opérationnelle immédiate donnée à LISP ne correspondait pas exactement à l'idée qu'on se faisait du langage initialement; par exemple, le lien entre les identificateurs et leur valeur est exprimé par une liste de paires construite dynamiquement au cours des appels de fonctions par liaison entre paramètres formels et effectifs; cette technique a été adopteée parce qu'elle est la plus simple à décrire opérationnellement (la récursivité en particulier se traite sans complication comme on l'a vu dans la section 1.1.3 de la première partie). Cependant, la liaison dynamique rend le sens des fonctions dépendant de leur contexte d'utilisation ce qui pose des problèmes évoqués plus loin (section 2.2.1) et en tous cas ne s'accommode pas d'une sémantique dénotationnelle simple qui indiqueraient naturellement une liaison statique. Remarquons toutefois que des sémantiques opérationnelles ont été données pour LISP ([SUSSMAN 81]) et ISWIM ([LANDIN 64]) avec une liaison statique mais elles sont forcément moins simples car elles doivent manipuler des piles d'environnements et des couples (expression, environnement) pour les définitions de fonctions.

2.1.3) Sémantique axiomatique.

Mentionnons également ce mode de description qui consiste à définir un langage par un ensemble d'axiomes valides sur ses constructions ([HOARE 69]). Ainsi la séparation est bien établie entre la base du langage qui est exprimée par ses axiomes et la partie correspondant à des choix de mise en oeuvre

précis. Notons de plus que des preuves de programmes peuvent être effectuées partir de ces axiomes et de règles d'inférence. L'ensemble des lois d programmes de FP, par exemple pourrait être considéré comme sa sémantiqu axiomatique plutôt qu'une conséquence de sa définition opérationnell ([WILLIAMS 82]).

2.2) PROBLEMES SEMANTIQUES.

Nous avons examiné jusqu'ici les moyens de décrire le sens des programme fonctionnels, plutôt que ce sens lui-même; comme on pouvait s'y attendre c'es principalement à propos des fonctions que la sémantique formelle doit apporte des éclaircissements; par exemple comment s'établit la correspondance entre le identificateurs et leur valeur, quel est le sens exact d'une fonction défini récursivement, que se passe-t-il quand un argument est indéfini? Voyon maintenant quelles réponses les langages fonctionnels ont fournies à ce questions.

2.2.1) Liaison statique ou dynamique.

Cette question a été soulevée précédemment à propos de LISP; l'exemple montre comment le sens d'une expression dépend du type de liaison adopté.

Exemple 2.
Quel est le résultat de l'évaluation de l'expression LISP suivante ([ALLE 78])?

```
\[[z];
    \ [[u];
       \ [[z]; u[B]] [C]]
     \ [[x]; cons[x;z]]]]
 [A]
```

- si une liaison statique est indiquée, l'identificateur z apparaissant dans cons[x;z] sera lié à A puisque de façon lexicale il correspond à la variable par la lambda-expression principale: le résultat sera donc (B.A).

- si le lien est effectué dynamiquement il en va autrement puisque dans le corps de l'expression \[[z];u[B]][C], z est lié à C et c'est dans ce nouvel environnement que le cons[x;y] sera évalué rendant alors (B.C).

Si la liaison dynamique facilite l'interprétation, en particulier la gestion des objets apparaîssant avant d'être déclarés, elle présente de graves inconvénients; tout d'abord les contrôles statiques possibles sont très réduits; plus grave encore le sens d'une fonction est désormais dépendant des noms des identificateurs qu'elle utilise. De plus, pour comprendre le sens d'un programme il ne suffit pas de connaître l'effet des sous-programmes mais il faut savoir comment ils sont réalisés; la décomposition des programmes de façon modulaire devient alors difficile à mettre en oeuvre; l'exemple suivant en donne un bon aperçu ([SUSSMAN 81]).

Exemple 3.

On utilise la fonction mapcar définie dans l'exemple 4.

```
scale[X] = * [X;C]
```

scale multiplie son argument par la valeur de C qui est dans la définition une

variable libre.

scalar-multiply[V;C] = mapcar[scale[V]]

scalar-multiply[(2,3,4),2] = (4,6,8)

puisque la liaison est réalisée de façon dynamique entre l'identificateur C et la valeur 2 lors de l'appel initial à scalar-multiply, elle reste valable dans le corps de scale qui est appelée ensuite; mais ce principe suppose de la part de l'utilisateur de scale une connaissance parfaite de la façon dont cette fonction est écrite car avec la définition suivante:

scale[X] = * [X;D]

le résultat n'est pas défini car D n'est pas lié lors de l'appel à scale. Ces problèmes ne se posent pas si la liaison est statique car on doit écrire:

scalar-multiply[V;C] = mapcar[\[[X]; * [X;C]];V]

Toutes ces raisons rendent la liaison statique plus satisfaisante en ce qui concerne la conception et la compréhension des programmes et la liaison dynamique est désormais proscrite ([LANDIN 64], [SUSSMAN 81]).

2.2.2) Fonctions récursives.

Soit la définition récursive suivante:

Exemple 4.

f(x,y) = si x = 0 alors 0 sinon f(x-1,f(x,y))

Si le sens d'une telle définition ne laisse guère de doute en mathématiques il en va autrement en informatique; toute la différence vient du fait qu'en informatique il est nécessaire de disposer d'une méthode de calcul de la fonction pour tout argument et qu'il faut prévoir le cas où celui-ci ne fait pas partie du domaine de définition. Or la formule de l'exemple 4 est plus une

équation qu'une définition et elle n'indique pas de moyen d'évaluer effectivement f. Il faut résoudre deux difficultés pour manipuler sainement de telles définitions:

- leur donner une signification de façon non équivoque
- trouver une mise en oeuvre compatible avec cette signification; cette question est traitée dans le chapitre 4 consacré à la mise en oeuvre et nous nous interessont ici à la définition de la sémantique.

Différents critères, parfois antagonistes, interviennent lors du choix d'une sémantique:

(1) Elle doit être générale: à toute équation récursive doit correspondre de façon non ambigüe une fonction entre deux domaines et celle-ci doit toujours être calculable (faute de quoi elle ne serait pas d'une grande utilité).

(2) Elle doit donner aux programmes un sens qui soit aussi proche que possible de l'idée intuitive qu'on s'en fait de manière à rendre leur écriture naturelle.

(3) Elle doit engendrer de puissantes méthodes de raisonnements sur les programmes.

(4) Elle doit permettre une mise en oeuvre efficace.

Les sémantiques opérationnelles indiquent directement un moyen de calculer le résultat et le plus souvent elles utilisent l'interprétation la plus naturelle: elles vérifient donc les critères (1) et (4) mais c'est généralement

au détriment des deux autres. Les preuves sont forcément liées à la notion d'exécution et donc assez lourdes tandis que le sens attribué aux programmes est parfois plus restrictif que l'idée qu'on s'en fait. Les définitions de Mac Carthy ([MACCARTHY 60]) et Landin ([LANDIN 66]) impliquent que la fonction f de l'exemple 4 soit indéfinie pour les arguments (1,1) alors qu'on s'attendrait au résultat 0. Ceci vient du fait que lors de leur interprétation, les arguments d'une fonction sont toujours évalués avant l'appel, ce qui conduit ici à demander sans cesse f(1,1): on appelle ce procédé l'appel par valeur. Notons cependant qu'il est toujours possible comme l'a fait Burge ([BURGE 76]) d'adapter la définition opérationnelle à l'appel par nom, où les arguments non évalués sont passés à la fonction avec l'environnement mais c'est au prix de la simplicité et de l'efficacité.

Le moyen de définir dénotationnellement la sémantique d'une équation récursive f = E(f) est de la considérer comme une transformation de fonctions T(f) = E(f) possédant des points fixes c'est-à-dire des fonctions pf vérifiant T(pf) = pf. La fonction représentée par l'équation doit alors être le moins défini de ces points fixes. Cette sémantique respecte les critères (1) et (3) puisque le point fixe minimal d'une transformation monotone existe toujours, est unique et est calculable. Il invalide généralement le simple appel par valeur mais des méthodes efficaces de mise en oeuvre lui ont été trouvées (elles font l'objet de la quatrième partie). De plus cette sémantique est la base théorique de nombreuses techniques de preuves. Le sens qu'elle donne aux programmes se rapproche suffisamment du concept intuitif, bien qu'il en diffère dans certains cas comme dans l'exemple suivant:

Exemple 5.

F(x) = si F(x) = 0 alors 0 sinon 0

Le point fixe minimal de la transformation de l'exemple 5 est la fonction indéfinie pour tout élément alors qu'on pourrait penser que cette équation définit la fonction constante 0 puisque c'est le seul résultat possible de la conditionnelle.

Manna utilise cet argument pour proposer un autre point fixe ([MANNA 77]), le point fixe optimal, qui exploite le maximum de renseignements contenus dans l'équation; par exemple dans le cas précédent le fait que les deux alternatives délivrent le même résultat est ignoré par le point fixe minimal mais pas par le point fixe optimal qui est la fonction constante 0. Celui-ci est défini grossièrement comme le plus grand point fixe compatible avec tous les autres points fixes (pour tout argument il rend la même valeur qu'un point fixe défini ou il n'en rend pas). Le principal inconvénient de cette méthode est cependant que le point fixe optimal est souvent difficile à évaluer, voire non calculable, ce qui constitue un handicap non négligeable. Le seul point fixe utilisé est le point fixe minimal et c'est donc lui que nous considèrerons à présent.

2.2.3) Fonctions strictes ou non strictes.

Si une fonction f est stricte en son ième argument, elle n'est pas définie quand il ne l'est pas; cette propriété s'exprime par la formule:

f(x1,...,xi,⊥,xi+2,...,xn) = ⊥

On dit qu'une fonction est stricte si elle l'est en tout argument. Il va de soi que si un système est défini entièrement à partir de fonctions et qu'elles

sont toutes strictes, les définitions récursives sont sans intérêt puisqu'elles calculent forcément ⊥. Il existe donc une construction fondamentalement non stricte dans tous les langages: la conditionnelle. En effet il est généralement admis que si

f(x) = si x = 0 alors 1 sinon 1/x

f(0) rend 1 bien que 1/0 soit indéfini. La conditionnelle peut être représentée par une fonction non stricte ([VUILLEMIN 74]), une expression spécifique ([MACCARTHY 60]), ou encore une forme fonctionnelle ([BACKUS 78]); cette dernière approche a l'avantage de permettre de façon élégante de maintenir toutes les fonctions strictes. Ainsi Burge traduit la conditionnelle

si P alors A sinon B

par l'expression:

[[if P][\[[];A];\[[];B]][]

ou les lambda-expressions \[[];A] et \[[];B] ne sont évaluées (alternativement) qu'au moment de leur application à la liste vide et rendent alors A ou B. La primitive if délivre donc une fonction d'ordre supérieur acceptant deux fonctions en argument; ce subterfuge est nécessaire si on veut éviter des fonctions non strictes; si les arguments sont directement les alternatives de la conditionnelle on écrira [if P][A;B] en désirant que [if P][A;⊥] rende A; le passage des arguments sous forme de fonction est donc un moyen de retarder, voire d'éviter leur évaluation puiqu'on ne calcule évidemment pas un tel objet infini avant de le transmettre mais seulement au moment de son application à un élément précis. On peut se demander quels sont les arguments en faveur (ou en défaveur) du maintien des fonctions strictes. Le problème se pose, comme

d'habitude sous trois aspects:

- l'aspect mise en oeuvre: ce n'est pas sans raisons, si les premières propositions ont toutes été strictes; le type d'appel classique (par valeur) consistant dans l'évaluation de tous les arguments avant leur transmission n'est pas adapté à une sémantique non stricte puisque dans ce cas l'un d'eux peut diverger sans que la fonction en soit affectée, ce qui signifie qu'elle ne l'utilisera pas systématiquement; il faut donc faire en sorte que leur évaluation soit retardée jusqu'au moment de leur utilisation; diverses techniques existent qui sont étudiées dans la quatrième partie.

- l'aspect souplesse d'utilisation; la fonction de l'exemple 4 diverge dans la plupart des cas si on lui impose d'être stricte mais converge partout sinon. L'utilisation de fonctions non strictes permet une écriture des programmes, immédiate, naïve, plus proche de la notation mathématique; on se soucie moins des deetails de mise en oeuvre car on sait que si le résultat existe intuitivement il sera calculé.

- l'aspect sémantique: la définition des primitives de façon non stricte et la possibilité pour les fonctions définies de ne pas l'être non plus influencent évidemment la signification donnée aux programmes.

Exemple 6.

```
Def f = eq0 -> 0; * o [f o sub1, f o add1]
Def eq0 = eq o [id,"0"]
Def sub1 = - o [id,"1"]
Def add1 = + o [id,"1"]
```

Comme on l'a vu, en FP toutes les fonctions sont strictes, de même que tous le combinateurs sauf la conditionnelle. Le point fixe mininal de la définitic récursive de l'exemple 6 est donc:

f1 = eq0 -> "0" ; "⊥"

Si maintenant on considère la construction et la primitive * non strictes savoir:

*: <0,⊥> = *: <⊥,0> = 0

(c'est la seule possibilité pour maintenir la continuité de la transformatio associée à l'équation).

Le point fixe minimal devient:

f2 = "0"

Remarquons que les particularités de FP, à savoir fonctions strictes à un seul argument et un combinateur de construction strict, lui permettent d'accomoder un appel par valeur à la sémantique du point fixe ([WILLIAMS 82]) qui n'est généralement pas valide dans ce cas ([CADIOU 72], [VUILLEMIN 74]). En effet, la fonction de l'exemple 4 s'écrit en FP:

Def f = eq0 o 1 -> "0"; f o [sub1 o 1,f]

et admet comme point fixe minimum:

f1 = eq0 o 1 -> "0" ; "⊥" en effet

T(f) = eq0 o 1 -> "0" ; f o [sub1 o 1,f]

T(f1) = eq0 o 1 -> "0" ;

(eq0 o 1 -> "0" ; "⊥") o [sub1 o 1, (eq0 o 1 -> "0" ; "⊥")]

T(f1) = eq0 o 1 -> "0"; (eq0 o 1 -> "0";"⊥") o [sub1 o 1, "⊥"]

T(f1) = eq0 o 1 -> "0"; (eq0 o 1 -> "0"; "⊥") o "⊥"

puisque la construction est stricte ce qui donne

T(f1) = eq0 o 1 -> "0"; "⊥" = f1

et la sémantique opérationnelle simple donnée par Backus ([BACKUS 78]) indique effectivement cette solution; on peut vérifier qu'au contraire la fonction

f1(x,y) = si x = 0 alors 0 sinon ⊥

n'est pas un point fixe de l'équation de l'exemple 4.

2.2.4) Les entrées/sorties.

Ce n'est pas sans raison si nous avons pu parler des langages fonctionnels jusqu'ici sans jamais évoquer les E/S; la réalité est qu'elles sont souvent réduites au minimum: la seule entrée est une expression à évaluer et la seule sortie le résultat de cette expression (FP, SASL, VAL,...). En fait ces restrictions draconiennes trouvent leurs causes dans les caractéristiques mêmes des langages fonctionnels:

1) Les fonctions n'ont pas d'autre effet que de rendre leur résultat et ne produisent pas de modifications d'un état global. Au contraire un ordre de sortie du genre "écrire" apparaît comme un effet de bord caractérisé et ne peut donc pas être introduit comme une primitive quelconque du langage sous peine de violer son caractère fonctionnel. Une fonction comme "lire" n'est pas banale non plus puisque, si on la combine, son sens ne dépend pas

uniquement de ses paramètres mais de son contexte d'exécution. Celui-ci reflètant lui-même les exécutions antérieures de "écrire", on voit que le style de programmation qu'ils permettent sort du cadre fonctionnel.

2) Ils n'imposent pas de schéma de contrôle précis: ceci ne pose pas de problème quand les effets de bord n'existent pas et que les sous-expressions peuvent être exécutées dans des ordres différents mais quand des entrées-sorties sont impliquées, qu'en résulte-t-il? Si l'enchaînement des opérations est inconnu à priori la programmation des E/S risque d'être périlleuse puisqu'on ne sait pas dans quel ordre ces actions seront effectuées ([FRIEDMAN 76]); par contre si le contrôle est explicité c'est une qualité importante des langages fonctionnels qui disparaît.

Jusqu'à présent, le problème des entrées-sorties a souvent été négligé dans la programmation fonctionnelle; souvent elles sont inexistantes (FP, SASL,...) ou alors elles ont été introduites sous forme de verrues sur le langage comme sur les systèmes LISP commercialisés par exemple. Ce n'est pourtant pas la meilleure façon de promouvoir les langages fonctionnels et on peut penser que cette question recevra désormais une attention accrue.

3) PREUVES ET TRANSFORMATIONS DE PROGRAMMES FONCTIONNELS.

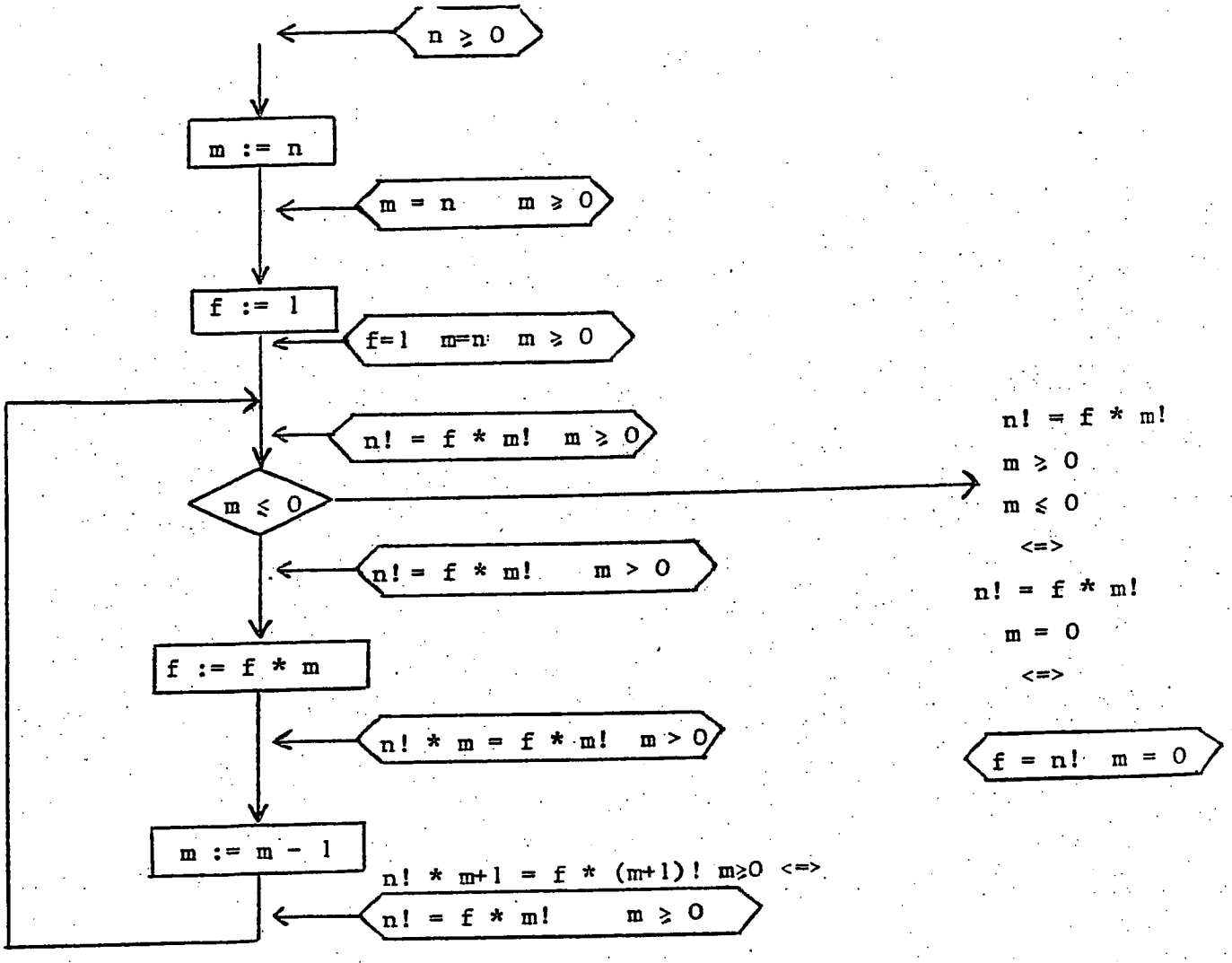
Le simple fait que tous les systèmes de preuves ou de transformations de programmes d'un certain intérêt pratique travaillent à partir de langages fonctionnels constitue un argument décisif en faveur de ce genre de programmation. En effet à l'heure où les coûts de mise au point et de maintenance prennent des proportions inquiétantes, la possibilité de vérifier de façon formelle les propriétés des programmes, ou de les construire de façon plus ou moins mécanisée n'est plus un luxe mais un besoin urgent.

Comparons en guise d'exemple la preuve d'un même programme dans deux langages: l'un impératif, l'autre fonctionnel.

Exemple 7.
Une expression itérative de factorielle dans une notation Algol pourrait être la suivante:

```
m:= n
f:= 1
tantque m > 0
   faire
       f:= f * m
       m:= m - 1
   fait
```

Montrons que ce programme calcule effectivement n! par la méthode des assertions inductives ([FLOYD 67]). cette méthode consiste à introduire des assertions entre chaque instruction du programme; la première représente les conditions d'entrée et la dernière doit impliquer la propriété souhaitée du programme. Les assertions intermédiaires représentent les propriétés valables apres l'exécution de l'instruction précédente au vu de son assertion d'entrée. On peut ainsi propager les assertions de part en part du programme (en commençant par le but ou le départ) pourvu qu'on ait une sémantique précise de chaque instruction. Il se pose cependant un problème dans le cas de la boucle puisqu'une assertion d'entrée depend elle-même de l'assertion de sortie de la boucle qu'elle permet de determiner; il faut donc introduire une assertion par boucle ce qui donne pour notre programme mis sous forme d'organigramme:


n ≥ 0
m := n
m = n m ≥ 0
f := 1
f=1 m=n m ≥ 0
n! = f * m! m ≥ 0
m ≤ 0
n! = f * m! m > 0
f := f * m
n! * m = f * m! m > 0
m := m - 1
n! * m+1 = f * (m+1)! m≥0 <=>
n! = f * m! m ≥ 0
n! = f * m!
m ≥ 0
m ≤ 0
<=>
n! = f * m!
m = 0
<=>
f = n! m = 0

La seule hypothèse est donc ici n>=0 et l'assertion de terminaison implique effectivement f = n!.

La difficulté de mécanisation provient des assertions de boucle à rajouter: elles doivent vérifier une propriété valable à chaque passage et de plus entraîner le résultat souhaité en sortie de boucle. Elle doit donc représenter une propriété statique (invariante) d'une structure essentiellement dynamique (la boucle), ce qui le rend parfois complexe (pour ce simple exemple n! = f * m! ne se déduisait pas de façon immédiate). Remarquons aussi que seule la preuve partielle est réalisée c'est-à-dire que si le programme se termine, il vérifiera les conditions voulues (d'autres techniques existent pour la terminaison).

Examinons les éléments qui rendent cette preuve laborieuse:

- La manipulation des variables: les assertions doivent manipuler toutes les variables utilisées; inutile de s'attarder sur les conséquences désastreuses dans le cas de programmes plus volumineux qui en manipulent un nombre important.

- Le contrôle explicite qui rend nécessaire un raisonnement au pas à pas, pour chaque instruction, ce qui n'est pas fait pour alléger la preuve.

Il n'est pas surprenant de constater que ce sont ces mêmes éléments qui compliquent l'écriture, la lecture et la sémantique des programmes.

Exemple 8.

En LISP, factorielle s'écrit:

```
fact[x]=[eq[x;0] -> 1; T -> * [x,fact[pred[x]]]
```

Si on suppose:

pred[succ[x]] = x et eq[succ[x],0] = F

x etant un entier positif,

fact[succ[x]]=[eq[succ[x];0->1;T->*[succ[x],fact[pred[succ[x]]

soit:

fact[succ[x]] = *[succ[x],fact[x]]

La comparaison avec l'exemple 7 se passe de commentaires; examinons plutôt quelques techniques de raisonnement adaptées aux langages fonctionnels et leur utilisation pratique.

3.1) METHODES DE PREUVES INDUCTIVES.

Ces méthodes sont basées sur la même idée que la récurence en mathématiques. Quand on désire prouver une propriété d'un programme sur un certain domaine, on peut le faire par induction sur le programme (induction dite du point fixe), sur le domaine de données (induction structurelle), ou sur la propriété ([MANNA 77]). Cette dernière possibilité est toutefois peu utilisée car il faut d'abord que la propriété s'y prête.

3.1.1) Induction de Scott (ou du point fixe).

Le principe est le suivant ([MANNA 73]): pour prouver une propriété P sur une fonction définie par l'équation récursive,

F = T(F)

, il suffit de montrer:

P(⊥) et ∀ F (P(F) => P(T(F))).

(⊥ etant la valeur "indefini")

On qualifie cette induction de "point fixe" car elle est basée sur la méthode de calcul du point fixe qui est la limite de $T(\underline{\lfloor})$. Ce type de transformation est donc naturellement adapté aux langages fonctionnels puisqu'il manipule des équations récursives; Il n'est cependant pas valide pour toute propriété P; la propriété

$$P1(F) = (F \text{ n'est pas complètement définie})$$

,par exemple, ne peut être démontrée de cette façon car elle peut être vraie pour toute fonction Ti(F) mais fausse pour la limite. On sait cependant que cette méthode est correcte pour les propriétés qui s'écrivent $P(F) = A(F) \underline{C} B(F)$ où A et B sont des transformations continues et l'inégalité $\underline{C}$ symbolise la relation "est moins définie que". Notons également qu'une propriété fausse pour $\underline{\lfloor}$ ne peut évidemment pas être montrée par cette méthode même si elle est vraie pour la fonction définie par le point fixe.

Exemple 9 ([MANNA 73])

$$F(x) = \text{if } p(x) \text{ then } x \text{ else } F(F(h(x)))$$

Montrons que le point fixe (minimal) pf de cette transformation T vérifie pf(pf)=pf à l'aide de la propriété P suivante:

$$P(F) = (pf(F) = F).$$

- $pf(\text{"}\underline{\lfloor}\text{"})(x) = pf(\underline{\lfloor})$

$= \text{if } p(\underline{\lfloor}) \text{ then } \underline{\lfloor} \text{ else } pf(pf(h(\underline{\lfloor})))$

par définition du point fixe

$= \underline{\lfloor}$

par définition de la conditionnelle

$= \text{"}\underline{\lfloor}\text{"}(x)$

par définition de "$\underline{\lfloor}$".

donc pf ("⊥") = ("⊥")

- supposons pf (F) = F

pf (T(F))(x) = pf(if p(x) then x else F(F(h(x)))

= if p(x) then pf(x) else pf(F(F(h(x))))

car pf("⊥") = "⊥"

= if p(x) then x else pf(F(F(h(x))))

car, par définition du point fixe,

pf(x) = if p(x) then x else pf(pf(h(x)))

donc if p(x) then pf(x) = if p(x) then x

et

pf (T(F))(x) = if p(x) then x else F(F(h(x)))

par hypothèse d'induction

= T(F)(x)

donc P(F) => P(T(F)) cqfd.

puisque cette propriété peut se mettre sous la forme:

pf F $\underline{C}$ F et F $\underline{C}$ pf F

la démonstration de P(pf) = (pf(pf) = pf) est donc achevée.

Cette méthode permet également de montrer des propriétés de systèmes de fonctions récursives et même de fonctions définies séparément en réalisant l'induction parallèlement sur chacune d'elles. Elle est donc puissante mais des problèmes se posent pour les preuves de terminaison: il est en effet impossible

de montrer P(F) = (h $\underline{C}$ F), h étant une fonction quelconque, puisque h $\underline{C}$ "[" n'est jamais vraie.

3.1.2) Induction structurelle.

C'est une généralisation de la récurence sur les entiers applicable sur des domaines D bien fondés, c'est-à-dire munis d'une relation d'ordre partielle < et ne possédant pas de chaîne décroissante infinie. Le principe est le suivant ([BURSTALL 69]); pour montrer une propriété P sur D il suffit de prouver:

$\forall$ a $\in$ D (($\forall$ b $\in$ D, b < a => P(b)) => P(a))

Dans la pratique, on doit montrer P de façon inconditionnelle pour le plus petit élément et on suppose l'existence d'un élément inférieur pour le cas général.

Exemple 10 ([MANNA 73]).

Le programme suivant calcule la "fonction 91":

f(x) = if x>100 then x-10 else f(f(x+11))

En effet, f(100) = f(f(111)) = f(101) = 91

f(99) = f(f(110)) = f(100) = 91

f(98) = f(f(109)) = f(99) = 91

Montrons f(x) = 91 pour tout x < 100.

On utilise pour cela l'ordre << suivant:

y<<x <=> x<y<100

Il est bien fondé et son élément minimum est 100.

- on a montré f(100) = 91.
- l'hypothèse de récurence est $\forall$y tq y<<x, f(y) = 91.

soit $\forall$y tq x<y<100, f(y) = 91

or f(x) = f(f(x+11))

si x+11>100 alors f(x+11) = x+1

et f(f(x+11)) = f(x+1)

or x+1 << x puisque x<x+1<100

donc f(x+1) = 91

par hypothèse de récurence

et f(x) = 91

si x+11 < 100 alors

x+11 << x puisque x<x+11<100

donc f(x+11) = 91

par hypothèse de récurence

et f(x) = 91

Cette méthode est donc très générale; elle est de plus, moins éloignée que la précédente de la logique du programme (l'ordre peu naturel qu'on est amené à trouver ici reflète sans doute la construction peu naturelle de cette fonction) et permet des preuves de terminaison.

3.2) METHODES SANS INDUCTION.

L'inconvénient des méthodes par induction est leur relative lourdeur; il faut faire la preuve à l'étape initiale puis à partir d'une étape la faire à l'étape suivante: ce raisonnement s'inspire plus de l'arithmétique que de l'informatique. D'autre part, ces techniques sont insuffisantes en elles-mêmes pour une automatisation des preuves puisque rien n'indique le moyen de raisonner à l'intérieur d'une étape (sans parler du choix de l'hypothèse de récurence (cf fonction 91); on est donc amené à examiner des méthodes de preuves sans

induction.

3.2.1) L'induction récursive.

Elle repose sur le principe suivant ([MACCARTHY 63]): deux fonctions qui vérifient la même équation récursive sont égales quand elles convergent. Cette affirmation, qui peut paraître naïve, permet en fait de montrer l'équivalence (partielle) de deux fonctions en les réécrivant sous une même forme récursive.

Exemple 11 ([LEROY 74]).

On suppose connues les primitives pred et succ sur les entiers et la propriété P: succ(pred(x)) = x, pred étant définie pour $x \neq 0$. Il est possible de réaliser l'addition par la fonction:

plus(x,y) = if y=0 then x else succ(plus(x,pred(y)))

On peut vérifier que plus(x,0) = x; montrons plus(0,x) = x en considérant les deux fonctions f(x) = plus(0,x) et g(x) = x.

- f(x) = plus(0,x) = if x = 0 then 0 else succ(plus(0,pred(x)))
  = if x = 0 then 0 else succ(f(pred(x)))
  par définition de f.

- g(x) = if x = 0 then 0 else x
  = if x = 0 then 0 else succ(pred(x))
  par P
  = if x = 0 then 0 else succ(g(pred(x)))
  par définition de g.

f et g se mettent donc sous la forme commune:

F(x) = if x = 0 else succ(F(pred(x)))

Ils sont donc équivalents (donc plus(0,x) = x) dans le domaine de convergence de cette définition, qui est ici l'ensemble des entiers (ce qui reste toutefois à montrer).

C'est donc une méthode naturelle, élégante et sans induction malgré son nom, les seuls problèmes venant du choix de la définition récursive commune, qui n'est pas toujours facile à faire, et de son domaine de convergence qu'il faut calculer d'une autre manière (sinon on peut réaliser des preuves illusoires car valides sur un domaine vide).

3.2.2) Le pliage-dépliage.

C'est une méthode de transformation de définitions récursives découverte par Manna ([MANNA 75]) et Burstall, Darlington ([BURSTALL 77]) et définie de la façon suivante:

- les objets manipulés sont des équations récursives,

- Les régles de transformations sont les suivantes:
  - la définition: c'est l'introduction d'une équation dont le membre gauche n'est pas un exemplaire de membre gauche d'une définition existante; par exemple:
  (1) f(x,y) = si x = 0 alors 1 sinon f(x-1,y)
  - l'instanciation: Elle consiste à réaliser des substitutions sur une équation. L'équation (1) peut s'instancier:
  (2) f(1,0) = si 1 = 0 alors 1 sinon f(1-1,0)
  - le dépliage: si un terme contient un exemplaire de membre gauche

d'équation on peut remplacer celui-ci par l'exemplaire du membre dr
correspondant: (2) peut se déplier en:

(3) f(1,0) = si 1 = 0 alors 1 sinon
(si 1-1=0 alors 1 sinon f((1-1)-1,0)

- le pliage: c'est l'opération inverse; (3) se plie en (2).

- l'abstraction: c'est l'introduction d'une clause where pour transform
une équation en utilisant des identificateurs auxiliaires. L'équati
suivante:

(4) g(x,y)=f(f(x,y),f(x,y)+1)

peut se transformer par abstraction en:

(5) g(x,y)=f(u,u+1) where u = f(x,y)

- l'application de lois sur les opérateurs de base: par exemp
l'application de lois sur l'égalité ((1=0)=F, (0=0)=T), sur la soustracti
((1-1)=0) et la conditionnelle ((si T alors x sinon y) = x,(si F alors
sinon y) = y). On transforme par exemple l'équation (3) en f(1,0) = 1.

Le système d'équations récursives est donc enrichi de façon répétitive pa
ces cinq règles.

Exemple 12.

Si la fonction append est définie en LISP par l'équation:

append[x;y] = [eq[x;nil] -> y ;
T -> cons[car[x];append[cdr[x],y]]]

et si

F[x;y;z] = append[cons[x;y];z]

On peut transformer cette équation en:

F[x;y;z] = eq[cons[x;y];nil] -> z

T -> cons[x;append[y;z]]

par dépliage de append et application des propriétés

car[cons[x;y]] = x et

cdr[cons[x;y]] = y

de la même façon eq[cons[x;y];nil] = F et

F[x;y;z] = cons[x;append[y;z]]

ce qui peut être considéré conne une nouvelle définition de F.

On obtient donc une méthode souple car il n'y a pas de schéma de transformation pré-établi mais qui a l'inconvénient de n'être que partiellement correcte: il faut donc prouver la correction d'une transformation ou trouver des conditions simples l'assurant ([KOTT 80]); remarquons de plus deux restrictions implicites que nous n'avons pas mentionnées jusqu'ici:

1) Cette technique ne peut transformer que des programmes fonctionnels puisque ses objets sont des équations récursives.

2) Elle est invalidée par un schéma d'appel par valeur qui, en forçant l'évaluation des arguments d'une fonction avant son invocation rend incorrect le simple dépliage de:

   f(x,y,z) = si x alors y sinon z

   La partie droite étant plus définie que la partie gauche, car les fonctions sont dans ce cas forcèment strictes.

Il se révèle de plus que les deux méthodes précédentes se complètent au sens où le pliage/dépliage peut être utilisé pour trouver la définition intermédiaire de la méthode de Mac Carthy. On obtient la méthode de pliage/dépliage généralisée ([KOTT 80]) en ajoutant aux lois sur les primitives

les propriétéss obtenues par la méthode d'induction récursive. Certaines dérivations impossibles à obtenir par pliage/dépliage deviennent alors réalisables comme le montre l'exemple 13.

Exemple 13.

Considérons les équations:

$G(n) = g(G(n))$

$H(n) = f(G(n))$

avec les propriétés sur f et g:

$f(g(n)) = g(g(f(n)))$

et $f(\underline{[}) = g(\underline{[})$

Ceci est un exemple (du à Milner) de cas où le principe de pliage/dépliage ne permet pas d'obtenir l'équation attendue c'est-à-dire $G(n) = H(n)$ mais la méthode généralisée y parvient ainsi:

$H(n) = f(G(n))$	définition
$H(n) = f(g(G(n)))$	dépliage
$H(n) = g(g(f(G(n))))$	propriété
$H(n) = g(g(H(n)))$	pliage
$G(n) = g(G(n))$	définition
$G(n) = g(g(G(n)))$	par dépliage

Le principe de Mac Carthy indique donc $G(n) = H(n)$.

3.3) MISE EN OEUVRE DE CES METHODES.

Aprés nous être étendus sur les méthodes théoriques de manipulation de programmes, il s'agit désormais d'étudier leur mise en pratique effective. Il est entendu que les programmes ne peuvent être manipulés manuellement quand ils deviennent importants, sous peine d'une perte d'intérêt pratique

évidente. Le problème se pose plutôt au niveau des choix à effectuer à chaque étape: doivent-ils être faits par la machine, et selon quel critère ou sont-ils laissés sous la gouverne de l'utilisateur? Les solutions radicales pêchent toutes deux; la première par un coût prohibitif et un manque de généralité, la seconde parce qu'elle se décharge sur l'utilisateur de la plus grande partie du travail. Les systèmes de transformation mis en oeuvre se situent donc généralement entre ces deux extrêmes.

3.3.1) Méthodes automatiques.

Le système de Boyer et Moore ([BOYER 75]) utilise un sous-ensemble de LISP et réalise automatiquement la preuve partielle de certaines propriétés de programmes qui sont exprimées elles même en LISP. Il part de l'idée que l'évaluation et l'induction se complètent car l'évaluation d'une fonction récursive décompose la structure de l'argument (jusqu'au cas initial) tandis que l'induction part du cas initial. Ainsi l'évaluation peut-elle être utilisée pour décomposer le cas général faisant apparaître le cas initial. Le procédé utilisé est le suivant: l'évaluateur LISP normal est d'abord invoqué (EVAL) qui rend son résultat, partiellement réduit (ou complètement dans les cas simples) dans une liste (BOMBLIST) qui est ensuite consultée pour savoir quelles expressions n'ont pas pu être calculées; une variable est alors choisie parmi celles-ci qui servira de variable d'induction; on appelle à nouveau EVAL pour calculer le cas initial (variable d'induction valant NIL), puis le cas général avec les hypothèses d'induction. D'autres techniques de base peuvent être utilisées pour débloquer les situations, à partir d'une heuristique pré-établie: ce

sont la normalisation et la réduction qui déduisent une expression sous une forme normale en utilisant les propriétés des opérateurs de base, la fertilisation qui fait intervenir l'hypothèse d'induction et surtout la généralisation qui remplace une formule par une variable quand le besoin s'en fait sentir.

Parmi les théorèmes prouvés ([BOYER 75]) citons:

l'equipotence de la fonction d'inversion d'une liste:

EQUAL[[REVERSE[REVERSE A]];A]

et les propriétés d'une fonction de tri:

ORDERED[SORT A]

IMPLIES[[EQUALS[SORT;A];A];[ORDERED A]]

Cependant, le système ne peut guère prouver de théorèmes plus compliqués que celui du tri; cela tient à la trop grande simplicité de son mécanisme d'induction (il n'est pas toujours facile de faire les bonnes hypothèses de départ) et du procédé de généralisation; car pour la faire de façon judicieuse il faut parfois choisir entre les occurrences d'une même variable et le choix n'est guère aisé.

Citons également le système automatique de Burstall et Darlington ([DARLINGTON 76]), qui a pour but d'améliorer l'efficacité d'un langage fonctionnel (POP-2 en l'occurrence) en le traduisant dans un langage impératif de la manière la plus intelligente possible; les moyens utilisés sont: l'élimination de la récursivité, la suppression des calculs redondants par des mises en facteur, le remplacment des appels de procédure par leur corps et la réduction de la taille mémoire nécessaire en imposant la réutilisation des cellules devenues inutiles. La récursivité est

d'abord éliminée à l'aide de règles de transformation de schémas récursifs en iteratifs avec leur conditions d'application (ces règles traitent un grand nombre de cas), d'un algorithme d'unification, d'un démonstrateur de théorèmes (pour prouver que les conditions d'application d'une règle sont vérifiées) et d'un programme de contrôle qui dirige le tout. Les autres étapes se déroulent systématiquement à partir d'une analyse fine du code pour détecter les relations entre les divers composants. Le système est donc orienté vers un but très précis: la traduction de programmes fonctionnels en programmes impératifs efficaces, c'est-à-dire proche de ce qu'un programmeur aurait directement écrit dans un langage traditionnel; l'avantage réside dans le fait que cette dernière version est plus obscure et certainement plus difficile à mettre au point que la première qui a été écrite sans souci d'efficacité.

Dans un état d'esprit tout à fait différent, le système FORMEL ([HUET 80]), permet également la réalisation automatique de preuves dans un langage proche de HOPE; le langage est caractérisé par un ensemble de constructeurs (par exemple NULL et CONS pour les listes) et un ensemble d'axiomes. L'algorithme de Knuth-Bendix est utilisé pour obtenir un système de règles de réécriture canonique, c'est à dire noethérien (un terme ne peut se dériver à l'infini) et confluent (un terme ne peut se dériver en deux termes différents et irréductibles). Il suffit alors pour montrer une propriété d'introduire cette nouvelle équation considérée comme règle de réécriture et de vérifier qu'elle forme avec les autres un système canonique. Le procédé possède donc l'avantage d'un raisonnement purement équationnel; son utilisation est très simple, le seul rôle de l'utilisateur etant d'indiquer éventuellement l'orientation d'une équation pour la

transformer en règle de réécriture. Le principal problème réside dans la terminaison de l'algorithme de complétion lui-même qui n'est pas assurée même si les orientations choisies sont correctes. Dans un tel cas l'utilisateur est contraint de recommencer entièrement la démonstration en modifiant le système de réécriture. De plus certaines preuves se traitent difficilement sous cette forme équationnelle (preuves par cas par exemple) et le système est amélioré par le meta-langage de preuves ML (décrit plus loin) qui permet une plus grande souplesse dans la conduite des preuves.

3.3.2) La méthode semi-automatique de Burstall et Darlington.

Elle met en oeuvre directement la théorie de pliage/dépliage décrite plus haut sur un langage fonctionnel assimilable à HOPE ([BURSTALL 80]) et dont les seules particularités qui nous intéressent ici sont:

- la définition des types de façon abstraite à partir de constructeurs produisant des sous-types disjoints (par exemple cons et nil pour les listes)

- la définition des fonctions sous forme de liste d'équations dont les membres gauches traitent le type de départ de façon exhaustive et disjointe, par exemple:

```
f(nil) <= 0
f(cons(a,b)) <= 1+f(b)
```

L'utilisateur soumet ses équations récursives au transformateur et attend les propositions du système qu'il peut accepter ou refuser. Le système n'est cependant pas complètement passif et il possède une stratégie pour appliquer les différentes règles de transformation qui est grosso-modo

la suivante ([BURSTALL 77]):

0: déplier toutes les équations tant que c'est possible

puis pour chaque "instanciation" d'équation à traiter:

1: déplier tant que c'est possible

2: appliquer arbitrairement une loi puis continuer par 1 ou 3.

3: réaliser un pliage après utilisation éventuelle de l'associativité, la commutativité et l'abstraction.

Le pliage est réalisé par un algorithme d'unification qui peut utiliser l'associativité, la commutativité et l'abstraction; ainsi ces propriétés sont utilisables seulement lorsqu'elles peuvent conduire à un pliage ce qui évite de les appliquer à tout moment sans résultat.

Le système fut conçu dans le but d'améliorer l'efficacité de programmes définis par des équations récursives; il s'est révélé que les règles bénéfiques de ce point de vue sont l'abstraction et l'application des propriétés des opérateurs; d'autre part, un bon procédé consiste à réaliser d'abord dépliages et applications de lois pour plier seulement en dernier lieu; ceci explique la stratégie donnée plus haut.

Cette méthode est adaptée en particulier à l'amélioration des programmes par élimination des tâches redondantes comme le souligne l'exemple suivant:

Exemple 14.

A partir de la définition habituelle de la fonction de Fibonacci qui se déroule en un temps exponentiel, on en dérive une solution qui la calcule en un temps linéaire [BURSTALL 77]; les étapes soulignées sont celles qui nécessitent une "trouvaille" de l'utilisateur, toutes les autres pouvant être produites automatiquement.

01	f(0) <= 1	définition
02	f(1) <= 1	définition
03	f(x+2) <= f(x+1)+f(x)	définition
04	g(x) <= <f(x+1),f(x)>	définition
05	g(0) <= <f(0+1),f(0)>	instanciation
06	g(0) <= <f(1),f(0)>	propriété de +
07	g(0) <= <1,1>	dépliage de 1 et 2
08	g(x+1) <= <f((x+1)+1),f(x+1)>	instanciation
09	g(x+1) <= <f(x+2),f(x+1)>	propriété de +
10	g(x+1) <= <f(x+1)+f(x),f(x+1)>	dépliage
11	g(x+1) <= <u+v,u> où <u,v>=<f(x+1),f(x)>	abstraction
12	g(x+1) <= <u+v,u> où <u,v>=g(x)	pliage
13	f(x+2) <= u+v ou <u,v>=<f(x+1),f(x)>	abstraction
14	f(x+2) <= u+v ou <u,v>=g(x)	pliage

On peut désormais définir f de la façon suivante:

f(0) = 1

f(1) = 1

f(x+2) = u+v où <u,v> = g(x)

avec g(0) = <1,1>

g(x+1)=<u+v,u> où <u,v> = g(x).

On obtient une nouvelle définition de fonction dès qu'on exhibe un ensemble d'équations dont les membres gauches couvrent son domaine de façon disjointe et exhaustive. Il y a de fortes chances pour que la nouvelle définition soit plus efficace (au sens de l'exécution sur une machine traditionnelle) que la première mais cette question n'a pas été formalisée.

On peut aussi par ce système réaliser la transformation de la récursivité en une forme "itérative". Puisque on manipule des équations récursives on ne peut pas produire directement une itération: il s'agit plutôt d'une récursivité à droite c'est à dire une équation: f(x1,...,xn)=E où E ne contient pas f ou est de la forme f(E1,...,En), les Ei ne contenant pas f.

Exemple 15.

Le but est de produire une forme "itérative" de factorielle

01	fact(0) <= 1	définition
02	fact(n+1) <= (n+1)*fact(n)	définition
03	f(n,u) <= u*fact(n)	définition
04	f(0,u) <= u*fact(0)	instanciation
05	f(0,u) <= u*1	dépliage
06	f(0,u) <= u	propriété de *
07	f(n+1,u) <= u*fact(n+1)	instanciation
08	f(n+1,u) <= u*((n+1)*fact(n))	dépliage
09	f(n+1,u) <= (u*(n+1))*fact(n)	associativité de *
10	f(n+1,u) <= f(n,u*(n+1))	pliage

11 fact(n+1) <= f(n,n+1) pliage de 2

On obtient donc fact à partir de f qui est trivialement convertible en itération.

f(0,u) <= u

f(n+1,u) <= f(n,u*(n+1))

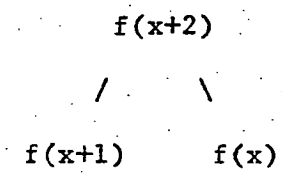
Notons que cette transformation est basée sur l'associativité de la multiplication.

Ces quelques exemples montrent les principaux points d'intervention de l'utilisateur à savoir:

- l'introduction de nouvelles définitions, comme celle de l'étape 4 de l'exemple 14 qui est capitale pour la suite du processus, ou la généralisation de définitions existantes comme dans l'étape 3 de l'exemple 15.

- l'instanciation judicieuse de ces définitions.

Ces interventions sont malheureusement celles qui contiennent l'idée de base pour optimiser la fonction (introduire une variable pour cumuler les résultats intermédiaires dans le cas récursif-itératif par exemple). Il serait donc intéressant de les supprimer; une solution serait d'étendre l'arbre d'exécution généré par une équation et de rechercher une unification possible des noeuds inférieurs et supérieurs qui symbolisera en quelque sorte une tranche de calcul répétitif; par exemple pour Fibonacci:



La substitution S(x)=x+1 transforme les noeuds inférieurs f(x+1) et f(x) en f(x+1) et f(x+2) (donc recoupement) ce qui suggère la définition g(x)=<f(x+1),f(x)> puisque f(x+2) s'exprime directement à partir de g(x) et g(x+1) s'écrit également à partir de g(x).

On a donc là un système puissant puisqu'il a même permis la synthèse de programme à partir de spécifications (non exécutables) ([DARLINGTON 81a]). Il constitue un équilibre entre une efficacité raisonnable et une certaine facilité d'utilisation. Notons cependant qu'il est vite submergé lors de transformations de plus grande envergure et qu'il laisse encore à l'utilisateur la conduite de certaines étapes clef de sorte qu'on peut dire que sa puissance est très dépendante de celle de l'utilisateur!

3.3.3) Les méta-langages de preuve.

Une autre approche consiste à laisser l'utilisateur libre de choisir ses propres stratégies mais de lui permettre de les programmer de façon à donner au système la responsabilité de les mener à bien; un avantage capital est alors la sécurité dans les manipulations et un confort appréciable quand les programmes à traiter deviennent volumineux. Il est donc nécessaire dans ce cas de disposer d'un méta-langage d'écriture des preuves sur le langage considéré. Un premier système réalisé dans cette optique est celui de M. Feather ([FEATHER 82]), ZAP, qui utilise également la technique de pliage/dépliage sur le langage fonctionnel HOPE. Dans ce

cas l'utilisateur décrit la transformation en indiquant une séquence d'actions à effectuer; chaque action constitue une indication précise de la modification à réaliser, la seule liberté laissée au système étant concrétisée par une possibilité d'unification.

Exemple 16 ([FEATHER 82]).

Connaissant les fonctions

```
Sum(nil) <= 0
Sum(cons(N,NUMLIST)) <= N+Sum(NUMLIST)
```

et

```
Squares(nil) <= nil
Squares(cons(N,NUMLIST)) <= cons(N*N, Squares(NUMLIST))
```

on veut transformer la définition

```
Sumsquares(L) <= Sum(Squares(L))
```

de façon à remplacer les appels à Sum et Squares par un appel récursif à Sumsquares.

Le méta-programme suivant produit l'effet désiré:

```
CONTEXT
  UNFOLD Sumsquares Sum Squares
  USING  Sumsquares
  TRANSFORM
    GOAL Sumsquares(nil) <= 0
    GOAL Sumsquares(cons(N,L)) <= $$(N,Sumsquares(L))
  END
END
```

ce qui signifie qu'il faut réécrire Sumsquares de la façon indiquée par les

instructions GOAL en dépliant éventuellement les définitions suivant UNFOLD et en utilisant la fonction Sumsquares (USING) dans la définition finale. Le second but est donné sous forme d'une fonction de N et de l'appel récursif: c'est un moyen pour l'utilisateur d'alléger la notation en n'exprimant que ce qui l'intéresse lors de la transformation. Le système procède en dépliant ce qu'il est autorisé à déplier jusqu'à ce qu'il puisse unifier les deux membres du but. Quand le but ne contient pas de synbole $$ il s'agit seulement d'une vérification. Par exemple, le premier but est traité ainsi:

```
Sum(Squares(nil))        dépliage de Sumsquares
Sum(nil)                 dépliage de Squares
0                        dépliage de Sum
```

et le second:

```
Sumsquares(cons(N,L))     $$(N,Sumsquares(L))
        |                        |
Sum(Squares(cons(N,L))    $$(N,Sum(Squares(L)))   depliage de Sumsquares
        |                        |
Sum(cons(N*N,Squares(L)))        |                depliage de Squares
        |                        |
N*N + Sum(Squares(L))            |                depliage de Sum
        \                       /
        unification de $$ avec \ x y . (x*x)+y
```

On obtient donc la nouvelle définition:

```
Sumsquares(nil) <= 0
```

Sumsquares(cons(N,L)) <= N*N+Sumsquares(L)

Il existe quelques autres ordres, mais cet exemple reflète la façon de mener les transformations dans ZAP. Quelques remarques s'imposent: tout d'abord il ne s'agit pas à proprement parler d'un langage puisqu'il ne consiste qu'en un ensemble d'ordres sans possibilité de nommer, donc de combiner des transformations pour en former de nouvelles. C'est un grand inconvénient quand il s'agit de traiter des exemples conséquents ou le même type de démonstration est à refaire de nombreuses fois. De plus la faible puissance des modèles tolérés (à travers $$) ne permet pas une abstraction suffisante ce qui est également gênant dans des cas complexes; on peut donc dire, que le méta-langage est ici de bas-niveau ce qui oblige l'utilisateur à écrire ses tactiques de façon très détaillée. Signalons toutefois, à la décharge de ZAP, que ce sytème a été utilisé pour le développement de programmes importants comme un compilateur par exemple.

LCF (pour Logic for Computable Functions) et son méta-langage ML (pour Meta-Langage) constituent un autre exemple de programmation des preuves. Une description grossière de ML pourrait être la suivante: c'est un langage fonctionnel d'ordre supérieur (tout au moins dans son sous-ensemble le plus utilisé), possédant les notions de type, type abstrait (non algébrique) et d'exceptions (ou d'erreurs). Les types sont contrôlés statiquement: ceux qui sont spécifiés sont contrôlés, les autres sont générés. C'est donc un langage d'intérêt général, orienté vers la conduite de preuves seulement par ses primitives de génération et de récupération des cas d'erreurs. ML est de plus spécialement adapté aux démonstrations sur PPLAMBDA (pour Polymorphic Predicate lambda-calculus) qui est un calcul déductif adapté à

la formalisation du raisonnement sur les définitions récursives de fonctions. En fait, PPLAMBDA est un calcul de base à partir duquel nombre de théories peuvent être construites de manière arborescente par l'ajout de constantes, d'axiomes et de types. Les propriétés vraies dans la théorie mère le restent chez ses filles qui ne sont qu'une spécialisation du langage de base pour un domaine d'application particulier (par exemple les listes); on retrouve donc ici les idées chères à Landin ([LANDIN 66]). En réalité PPLAMBDA est construit en ML par l'intermédiaire des types abstraits: chaque classe syntaxique de PPLAMBDA est représentée par un type ML. Ainsi la classe des types PPLAMBDA correspond-elle au type "type" de ML! ML n'est spécialement adapté à PPLAMBDA que pour deux raisons:

- il est permis d'écrire les termes de ce langage sous une forme concrète PPLAMBDA plus manipulable car les types ML assurant l'interface avec PPLAMBDA sont prédéclarés et le traducteur est fourni de manière standard.

- un ensemble d'axiomes et de règles d'inférence de PPLAMBDA est également offert à l'utilisateur.

Si celui-ci n'est pas satisfait du calcul proposé, il peut en définir un autre en suivant la même démarche: ML allie généralité et facilité d'utilisation dans le cas standard.

Un premier outil de construction de preuves est constitué par les tactiques qui permettent de décomposer un but en sous-buts.

Le type tactic est prédéclaré en ML:

```
tactic = goal -> (goal list X proof)
```

où X représente le produit cartésien.

ce qui résume ce que nous venons de dire: une tactique rend à partir d'
but B1 une liste de buts B'1...B'n et une preuve P sensée montrer B1
partir de B'1...B'n. Le type proof est défini ainsi:

```
proof = thm list -> thm
```

Une preuve déduit donc un théorème d'une liste de théorèmes.
distinction entre le type formule (form) et le type théorème (thm) e
capitale car le contrôle de types ML assure qu'en aucun cas une formu
fausse dans une théorie ne peut en devenir un théorème. L'utilisateur pe
produire des tactiques invalides c'est-à-dire telles que la réalisation d
sous-buts n'implique pas celle du but: leur seul effet sera de produire
échec ou de démontrer un autre théorème que celui qu'on attend mais jama
de prouver un théorème incorrect. En effet, le seul moyen de produire
théorème est de le déduire de règles d'inférence valides dans la théorie
de l'imposer comme axiome (auquel cas il est bien sûr conseillé de ne p
introduire de contradiction); ceci est assuré par le contrôle de typ
Explicitons maintenant le type goal:

```
goal = form X simpact X form list
```

ou simpact est un ensemble de règles de simplification et form représen
les formules PPLAMBDA; le but est alors de prouver la première formule
partir de la liste de formules en utilisant éventuellement les règles
simplification.

Une tactique, pour être intéressante, doit produire à partir d'un bu
démontrable des sous-buts qui le soient aussi, faute de quoi elle pourrai
conduire à un échec alors que la preuve est effectivement possible. Un

tactique valide possédant cette qualité est dite fortement valide.

Il existe un certain nombre de tactiques fortement valides prédéfinies; en voici quelques exemples:

CONDCASESTAC: elle recherche une condition (valeur booléenne) libre dans la formule et produit les trois sous-buts correspondant à ses trois valeurs possibles (vrai, faux, indéfini); le principe utilisé est de montrer la propriété d'une conditionnelle en envisageant les trois cas possibles pour le test.

SIMPTAC: elle transforme le but initial en appliquant à la formule à montrer, les règles de l'ensemble de simplification.

GENTAC: elle supprime la quantification d'une formule en évitant les conflits entre identificateurs.

Il est également possible de construire des tactiques plus générales (les stratégies) à partir de celles de base à l'aide de combinateurs de tactiques qui préservent la validité forte. Citons par exemple:

- REPEAT: tactic -> tactic
  qui répète une tactique tant qu'elle est appliquable.

- ORELSE: tactic = tactic -> tactic
  qui applique la première tactique si elle est applicable, la seconde sinon.

- THEN: tactic = tactic -> tactic
  qui applique successivement les deux tactiques.

Exemple 17.

En PPLAMBDA la conditionelle se note A => B | C. Le but est de montrer le théorème suivant:

```
∀ P1, P2, X, Y, Z, W
     (P1 => (P2 => X | Y) | (P2 => Z | W))   ≅
     (P2 => (P1 => X | Z) | (P1 => Y | W))
```

Le procédé intuitif est celui-ci: libérons nous des quantificateurs et étudions au cas par cas selon les valeurs possibles des conditions en réalisant des simplifications quand c'est possible. Il suffit donc d'écrire en ML:

```
(REPEAT GENTAC) THEN (REPEAT (CONDCASESTAC THEN SIMPTAC))
```

LCF constitue donc un outil de construction de preuves de programmes très général à tout point de vue:

1) il n'est pas cantonné à un langage précis puisque nombre d'entre eux peuvent être axiomatisés comme des théories de PPLAMBDA. Un exemple est donné par la description de FP (étendu) de cette manière ([LESZCZYLOWSKI 80]). La démonstration des théorèmes de Backus ([BACKUS 78]) a été réalisée en ML à partir des axiomes introduits pour définir les primitives et les combinateurs de base;

2) il n'est pas spécialisé dans une logique de preuves spécifique: on peut réaliser les preuves par induction structurelle, induction du point fixe,...

3) il n'impose pas une manière précise de conduire les preuves, bien que préconisant une démarche descendante (buts vers sous-buts). On peut dire

réutilisé, éventuellement modifié, si des changements interviennent, pour produire une version satisfaisant les nouvelles conditions. On obtient des programmes compréhensibles par leur version initiale (si le langage utilisé est satisfaisant à ce point de vue), corrects et efficaces par la transformation (si on dispose de techniques suffisamment puissantes) et évolutifs par la réunion des deux. N'étaient-ce pas les buts que nous nous étions fixés dans l'introduction? Notons également en conclusion un autre intérêt des méta-langages constitué par la réunion possible d'un ensemble de stratégies d'intérêt général permettant d'augmenter nos connaissances au niveau de la conception des algorithmes.

que ML constitue un trait d'union entre les systèmes qui se contentent de vérifier des étapes de preuves élémentaires et ceux qui réalisent la démonstration entièrement automatiquement; en effet, si l'utilisateur dispose d'une méthode de preuve suffisante (par exemple dans le cas des problèmes connus ayant déjà fait l'objet d'études théoriques) il peut le programmer et laisser le système faire le reste; sinon il peut réaliser la démonstration étape par étape, le système vérifiant ses transformations; mais entre ces deux cas extrêmes, toutes les possibilités sont permises: des programmes partiels de preuves peuvent être utilisés dans les étapes, il y a en fait un passage continu du vérificateur de preuves au démonstrateur de théorèmes, la seule distinction étant la "grosseur" des étapes traitées.

L'utilisation de stratégies valables pour un nombre de programmes variés, faciles à comprendre et à modifier permet de franchir un nouveau pas dans la formalisation de la notion de construction de programmes. On peut imaginer une séparation quasiment totale en deux étapes correspondant chacune à une contrainte de la programmation à savoir: d'une part écrire des programmes justes et compréhensibles, d'autre part en produire une version exécutable et efficace (cette étape étant clairement dépendante du type de matériel supportant l'application). La version finale du programme peut rester complètement cachée puisqu'elle est assurée de correspondre aux spécifications initiales; il suffit de garder le programme initial et le méta-programme utilisé pour le dériver: ils sont respectivement garants de la logique et de l'efficacité, deux aspects généralement confondus dans les programmes classiques ce qui pose les problèmes connus lors de leur modification ou leur preuve. Ici au contraire le méta-programme peut être

3ème Partie:

ETUDE COMPARATIVE DES DIFFERENTS LANGAGES FONCTIONNELS EXISTANTS.

3ème Partie: Etude comparative des différents langages fonctionnels existants.

1) Principales différences existant entre les langages fonctionnels.

   1.1) Contrôles statiques.

   1.2) Variables.

   1.3) Combinateurs.

   1.4) Fonctions strictes ou non strictes.

   1.5) Where-expressions.

2) Tour d'horizon des langages existants.

Les langages fonctionnels ont été évoqués jusqu'ici à partir de leurs caractéristiques communes; il est cependant apparu, à la lumière des exemples choisis, qu'il existe parfois des différences notables entre eux (voir LISP et FP par exemple). Le but de ce chapitre est de détailler ces divergences, d'en étudier les conséquences et de montrer les choix qui ont été faits pour les principaux langages applicatifs connus.

1) PRINCIPALES DIFFERENCES EXISTANT ENTRE LES LANGAGES FONCTIONNELS

1.1) CONTROLES STATIQUES

On a pu constater que, parmi les langages cités en exemple, aucun d'eux ne possédait la notion de type; ceci implique une grande souplesse d'utilisation et simplifie la tâche d'écriture des programmes mais comporte cependant des inconvénients non négligeables; le premier est bien sûr la pauvreté des contrôles statiques qu'il est possible de mettre en oeuvre, puisqu'a priori les arguments des fonctions peuvent appartenir à un domaine quelconque. Le prix à payer est donc une vérification systématique du type à l'exécution, ce qui est une source d'inefficacité notoire. La facilité d'associer des domaines spécifiques aux fonctions permet dans certains cas de détecter des incohérences de manière statique, évitant ainsi une exécution sans espoir. Un autre avantage du typage réside dans la possibilité d'utiliser des types abstraits; on construit alors un nouveau type en séparant sa définition logique caractérisée par les fonctions qui peuvent le manipuler (et les relations qui existent entre elles) de son mise en oeuvre réelle à l'aide de types pré-existants (cas de ML, HOPE ..).

Les programmes utilisateurs sont alors complètement dégagés de la façon dont le type est effectivement représenté au niveau du langage, étant entendu que cette représentation doit vérifier les propriétés abstraites annoncées; on a de plus l'assurance que les données de ce type seront forcément utilisées à bon escient et qu'elles resteront donc cohérentes

grâce au contrôle statique. Le langage HOPE ([BURSTALL 80]) est exemplaire à ce sujet; les types abstraits sont définis à l'aide de modules dont les objets exportés sont le type défini et ses fonctions associées; certaines de ces fonctions sont privilégiées et appelées les constructeurs de type (par exemple nil et cons pour une liste); des fonctions utilitaires sont définies par autant d'équations que le type de leur argument possède de constructeur; une fonction sur les listes, par exemple, devra être définie par deux équations correspondant aux deux constructeurs possibles:

f(nil) <= E1 et

f(cons(x,y)) <= E2.

Ainsi le contrôle statique vérifie non seulement la cohérence des types mais aussi la complétude de la définition puisque tous les constructeurs doivent être envisagés. Un autre exemple remarquable est ML qui permet de concilier les avantages du typage à ceux de l'absence de type. En effet l'utilisateur peut associer un type à un objet dont il désire spécialement restreindre l'utilisation, ou ne pas le faire si bon lui semble. Dans ce cas le système déduira automatiquement le type de l'objet à partir de sa définition.

Exemple 1.

si f est définie par f(x) = 2*x;

ML attribue à f le type (int -> int), l'indique à l'utilisateur et l'utilise lors des vérifications ultérieures.

Une autre proposition dans ce sens est celle de Guttag ([GUTTAG 81]) qui suggère d'expliciter le contrôle de type et d'introduire les types abstraits dans FP. A chaque primitive p est associée une fonction booléenne

E(p) exprimant son domaine de définition ; par exemple:

E(+) = eq o [<INT,INT>,type]

cette fonction permet alors de remplacer la primitive p par une conditionnelle explicitant la séparation entre le contrôle préalable du type et le calcul même de la fonction:

p = E(p) -> p´; ⌊.

De la même façon il existe pour chaque combinateur un mode de calcul de son domaine à partir de ceux des fonctions auquelles il est appliqué, par exemple:

E(f o g) = E(g) -> (E(f) o g´-> T; F); F).

On peut ainsi éviter des contrôles inutiles à l´exécution en calculant pour chaque fonction définie F son domaine E(F) par les règles précédentes puis en transformant ces expressions à l´aide de l´algèbre de programmes afin d´éliminer les redondances.

Exemple 2.

Def max = >= -> 1;2.

Cette définition est transformée en:

Def max = isintpair -> (>= 1;2);⌊.

où les fonctions >=, 1 et 2 n´ont pas de contrôle à réaliser à l´exécution. Le test isintpair (paire d´entiers) assure la correction de leur argument.

Les types abstraits sont définis au moyen de spécifications algébriques en séparant les axiomes vérifiés par les opérateurs du type de leur mise en oeuvre dans les termes du langage, puis en vérifiant la cohérence de ces deux spécifications. Il est également souhaitable de

permettre la restriction des fonctions à certains types, par exemple:

```
def f(x:type1, y:type2) = g o [h o x,y]
```

Il s'agit alors de traduire automatiquement cette forme dans le langage de base, on obtient dans ce cas:

```
def f = type1 o 1 -> (type2 o 2 -> g o [h o 1,2]; 1); 1
```

En dernier lieu insistons sur l'intérêt des types polymorphiques (ML, ALFA, HOPE) ou types variables qui représentent en quelque sorte une classe de types supportant des traitements communs. Le type " liste de type1 " en est un exemple; si type1 identifie un type quelconque, liste de réels sera un élément de cette classe. On gagne ainsi en souplesse d'utilisation puisqu'une fonction n'admet pas forcément un type précis en argument mais un ensemble de types qu'elle peut manipuler de façon identique.

1.2) VARIABLES.

La caractéristique la plus frappante de FP réside dans l'absence totale de variable dans ce langage (on utilise le terme variable en tant que identificateur d'objet par opposition aux identificateurs de fonctions). Le problème posé par les fonctions à plusieurs arguments est résolu en considérant qu'ils sont regroupés dans une séquence et que la fonction les accède à l'aide de sélecteurs. Ainsi

$$f(x,y,z) = x+(y-z)$$

s'écrit en FP:

Def f = + o [1,- o [2,3]]

Backus [BACKUS 81] ne manque pas d'arguments pour justifier ce choix: tout d'abord il implique une programmation structurée puisque c'est seulement par combinaison de fonctions (à l'aide de combinateurs fixés) qu'on peut en construire de nouvelles. De plus le raisonnement au niveau des fonctions facilite les preuves de programmes, ainsi que le suggère l'exemple 3:

Exemple 3.

La loi suivante énoncée en LISP:

\[[y];[\[x];[f[x];g[x]]][h[y]]] =

\[[y];[\[[x];f[h[x]]][y],\[[x];g[h[x]][y]]

semble plus difficile à identifier qu'en FP:

[f,g] o h = [f o g, f o h]

On voit qu'ici la mention explicite des objets est inutile et ne fait qu'obscurcir la notation. D'autre part on peut penser que si une loi de

cette simplicité est difficile à reconnaître, les raisonnements manuels ou semi-automatiques risquent d'être difficiles à mettre en oeuvre. L'assimilation d'une liste d'arguments à une séquence permet d'autre part une plus grande souplesse dans la combinaison des fonctions en FP. En effet, une fonction prenant en argument une séquence de plusieurs éléments peut être combinée indifféremment avec une fonction rendant cette séquence entière ou un ensemble de fonctions rendant chacune un élément; ceci n'est pas permis dans le cas général car deux fonctions définies par:

f(x,y) = E et g(x,y) = (x,y)

ne peuvent être combinées directement en

f(g(x,y))

Si les variables sont généralement utilisées malgré ces critiques, c'est qu'elles facilitent dans la plupart des cas la lecture des définitions comme le montre l'exemple suivant.

Exemple 4.

Soit la fonction FP suivante définie à l'aide de variables:

(1) Def f: <x,<y,z>> = <<x,y>,<x,z>>

La définition suivante sans variables

(2) Def f = [[1,1o2],[1,2o2]]

est non seulement plus obscure que la première mais ne lui est pas non plus strictement équivalente puisqu'elle autorise l'application de f à une séquence de plus de deux éléments. Pour traduire fidèlement la première définition il faudrait écrire:

(3) Def f = eq o [lentgh,"2"] -> (eq o [length o 2,"2"] ->
[[1,1o2],[1,2o2]]; "[]"); "[]".

ce qui est une expression encore moins lisible et moins naturelle que la précédente.

Pour surmonter ces difficultés sans retomber dans les travers de la programmation au niveau de objets, Backus ([BACKUS 81]) propose une définition étendue prenant la forme d'une équation qui doit être vérifiée par la fonction définie. La définition étendue de f est:

(4) Def f o [g,[h,i]] = [[g,h],[g,i]]

Elle se rapproche donc beaucoup de la première (1), la plus naturelle, à la différence près qu'elle ne manipule pas d'objets; elle constitue une équation de fonctions valable quelles que soient g, h et i et resulte donc directement en une loi algébrique sur la fonction définie. Comme en (1) la restriction aux séquences de deux éléments s'exprime naturellement et la lisibilité est indiscutablement supérieure à la définition (3). Une méthode est également proposée qui permet de passer automatiquement de cette forme étendue à une expression en FP pur en tenant compte des restrictions imposées par l'équation d'origine (notons toutefois que la forme produite est loin d'être optimisée).

Il n'a jamais été évoqué dans ce débat le problème de la puissance d'expression. Toutes les fonctions définies à l'aide de variables peuvent elles être exprimées sans variables? La réponse est donnée par un résultat de mathématiques ([SCHONFINKEL 24]) qui indique que l'on peut toujours transformer une fonction de plusieurs arguments en une fonction d'ordre supérieur à un seul argument (ce qui ôte donc la nécessité des variables) en introduisant des constantes supplémentaires appelées combinateurs. Des traductions automatiques ont été réalisées. Citons celle de Burge pour le

lambda-calcul ([BURGE 76]) réalisée à l'aide de deux combinateurs seulement et de Turner pour son langage SASL ([TURNER 79]). Cette possibilité, outre son aspect théorique important, peut être utilisée en vue d'une mise en oeuvre plus efficace car les formules combinatoires obtenues, malgré leur taille imposante et leur manque de lisibilité s'adaptent très bien à une execution sur des structures de machines non conventionnelles ou à des technologies nouvelles comme les VLSI.

1.3) COMBINATEURS.

D'autres langages (LISP, ML,...) ne possèdent pas la notion de combinateur et les fonctions sont définies par des expressions dépendant de leurs arguments. L'utilisateur peut alors créer des fonctions d'ordre supérieur, c'est-à-dire ayant des fonctions en argument et en résultat, qui jouent le même rôle que des combinateurs (voir par exemple la définition de MAPCAR en LISP qui correspond à 0< de FP (section 1.1.4 de la première partie)). La possibilité de manipuler directement des fonctions est très importante dans les langages fonctionnels et il existe toujours l'un de ces deux moyens: combinateurs ou fonctions d'ordre supérieur (ALFA possède même les deux). Pouvoir se fabriquer ses propres opérateurs semble plus prometteur à première vue puisque l'on n'est pas limité à un ensemble restreint de formes fonctionnelles. Par contre les combinateurs favorisent une meilleure structuration des programmes. L'exemple 3 est à ce propos très révélateur. Cette critique est comparable à celle des caractéristiques impératives des langages conventionnels et Backus va même selon cette logique, jusqu'à considérer LISP comme le FORTRAN des langages fonctionnels ([BACKUS 81])! Cependant, il ne suffit pas de s'entendre sur l'intérêt des opérateurs, le plus difficile reste de choisir les bons. Ce choix est d'autant plus important s'il est impossible, pour l'utilisateur, de créer de nouveaux combinateurs. Si les formes fonctionnelles sont trop peu nombreuses ou trop peu expressives, la puissance du langage s'en ressent cruellement mais si à l'inverse elles sont trop puissantes ou abondantes c'est la lisibilité qui risque d'en souffrir. Cette remarque est souvent faite à propos d'APL ([IVERSON 62], [IVERSON 79]) dont certains opérateurs sont à la fois monadiques et dyadiques réalisant dans chaque cas

un traitement très différent et d'autres effectuent des opérations si complexes que leur manipulation est rendue très périlleuse. Si l'unanimité est faite sur un petit ensemble d'opérateurs (comprenant entre autres les formes Ø et / de FP), les propositions nouvelles ne manquent pas, signalons par exemple:

- l'opérateur tree de Williams ([WILLIAMS 81]) défini par:

  tree f:<x1,...,xn> = f:<tree f:<x1,...xk>,tree f:<xk+1,...,xn>>

  (avec k = n/2)

  qui est équivalent à / quand f est associative mais est plus satisfaisant dans le cas d'une interprétation parallèle.

- l'opérateur scan(\) de Iverson ([IVERSON 79]) tel que:

  \f:<x1,...,xn> = <x1,f:<x1,x2>,...,/f:<x1,...,xn>>

  soit par exemple: \+:<1,2,3> = <1,3,5>

- l'opérateur de compression APL noté /:

  1 0 1 / 4 3 2 = 4 2

  seuls les éléments de l'opérande droit correspondant à une expression booléenne valant vrai à gauche sont conservés. Cet opérateur apparaît ailleurs sous d'autres formes; par exemple:

  (filter odd):[1,2,3,4] = [1,3]

  dans [WADLER 81] qui filtre les éléments d'une séquence selon une condition, ou odd$square: <1,2,3,4> = <1,9> dans [CHIARINI 80] qui réalise un "apply to all" (Ø) sélectif.

Wadler ([WADLER 81]) a montré combien un nombre réduit d'opérateurs peut faciliter l'automatisation des transformations de programmes, en construisant un ensemble complet de règles de réécriture (toute

composition d'opérateurs peut se réduire en un seul opérateur) à partir de trois combinateurs de base. La définition obtenue à l'issue de la transformation contient un seul opérateur appliqué à une fonction composée de conditionnelles et de where-expressions. Elle est plus efficace que la définition initiale au sens où les parcours de listes y sont minimisés.

1.4) FONCTIONS STRICTES OU NON STRICTES

La définition et les arguments en faveur ou contre cette propriété ont déjà été évoqués dans le chapitre consacré à la sémantique (section 2.2.3) de la deuxième partie). Rappelons que la possibilité de définir des fonctions non strictes permet (mises à part les conséquences au niveau de la sémantique et de la mise en oeuvre) une plus grande souplesse dans l'expression des problèmes puisque des fonctions qui seraient indéfinies dans le cas général peuvent rendre un résultat dans le cas non strict. La manipulation de listes potentiellement infinies par exemple est tolérée à condition qu'il n'y ait pas de tentative d'y accéder dans leur totalité ([FRIEDMAN 78], [MORRIS 80], [TURNER 81b]). L'écriture des programmes peut donc être plus proche du langage mathématique et plus détachée des contraintes de mise en oeuvre. Plusieurs choix sont possibles dans le cas où on tolère les fonctions non strictes: celles-ci doivent-elles être spécifiées par le programmeur à l'aide d'un moyen quelconque, par exemple lcons pour lazy-cons au lieu de cons (strict) dans [BURSTALL 80]), ou est-ce qu'il s'agit d'une règle générale (SASL, POPLAR)? La première solution laisse plus d'initiative au programmeur qui doit juger lui-même de l'opportunité de rendre les

fonctions non strictes dans chaque cas; par contre la seconde est plus claire et plus souple au niveau de la mise en oeuvre. Cependant elle nécessite des techniques d'optimisation afin de ne pas implémenter systématiquement un appel par nom qui est souvent plus coûteux et n'est pas toujours nécessaire ([MYCROFT 80]).

1.5) WHERE-EXPRESSIONS

Ce sont des expressions de la forme:

<expression> where <définition>

avec <définition> ≡ <identificateur> = <expression>

Elle permettent donc de structurer les programmes avec les avantages que cela comporte (lisibilité,maintenabilité, possibilité de récupération d'espace mémoire) mais aussi les inconvénients (problèmes de liaisons entre les identificateurs et leurs valeurs). La plupart des langages fonctionnels connaissant les variables possèdent cette facilité, en particulier ceux qui sont inspirés du lambda-calcul ([LANDIN 64], [BURGE 76]) puisqu'une where-expression comme

E1 where x = E2

se traduit directement par:

\[[x];E1][E2]

Mentionnons également la let-expression qui est une variante de la where-expression s'écrivant: let x=E2 in E1.

2) TOUR D'HORIZON DES LANGAGES EXISTANTS.

Les langages fonctionnels les plus importants ont été regroupés dans le tableau 1 qui permet de constater les différents choix qui ont été effectués pour chacun d'eux, en rapport avec la discussion précédente. Chaque langage est accompagné de la référence qui a été utilisée, ce qui permet d'éviter les confusions pour ceux d'entre eux dont diverses variantes ont été définies ou implémentées. Le choix de ces langages peut bien sûr prêter à discussion en particulier pour ceux qui ne sont pas à proprement parler purement fonctionnels. APL est mentionné en raison de sa renommée et parce qu'il a été le premier à montrer l'intérêt des combinateurs, PROLOG en raison des nombreuses caractéristiques communes qu'il possède avec les langages fonctionnels (dans sa version pure)... D'autres auraient pu être cités, qui ont influencé les langages actuels comme GEDANKEN ([REYNOLDS 70]) qui est basé sur le principe de la complétude (toute valeur est permise dans tout contexte sensé) et traite les structures de données de manière fonctionnelle; parmi les propositions récentes CDS ([BERRY 81]) est basé sur la notion de structure de données concrète, ou cds, qui représente un ensemble de cellules pouvant contenir des valeurs atomiques selon certaines règles d'accessibilité; les propriétés mathématiques qui en découlent sont très puissantes puisque l'ensemble des algorithmes entre deux cds M1 et M2 est lui-même un cds M1 -> M2 et que la sémantique d'un algorithme est représentée par l'algorithme lui-même (deux algorithmes sont donc égaux

s'ils sont textuellement égaux)!

Il convient spécialement de mentionner le langage ML déja décrit plus haut et qui connait un développement très important. Rappelons qu'il a été influencé par ISWIM ([LANDIN 66]) et peut être considéré comme langage de programmation ou comme méta-langage de preuve dans le système LCF.

FFP ([BACKUS 78]) est un langage d'ordre supérieur pouvant être considéré comme une formalisation de FP. En effet, les objets de ce langage peuvent être des fonctions et une régle, la métacomposition, permet l'application de ces fonctions à un objet. Il est ainsi possible de construire en FFP les formes fonctionnelles de FP (0<, /,...).

La proposition de Friedman et Wise ([FRIEDMAN 80]) concerne une nouvelle structure de données, les multiensembles, construits par la primitive non déterministe "frons". Celle-ci correspond à un cons suspendu à la différence près que l'ordre d'un élément dans le multi-ensemble n'est déterminé qu'à partir du moment où il a été évalué. Quand un élément de la structure est demandé, tous les calculs sont lancés simultanément et le premier à converger constituera l'élément de tête du multi-ensemble; celui-ci devient alors hybride puisque la partie suspendue reste désordonnée. Un tel constructeur permet, par son non-déterminisme, de résoudre avec élégance certains problèmes. De plus, il est naturellement interprété de façon parallèle. Les problèmes apparaissent toutefois à propos du sens à lui attribuer en particulier combiné avec le constructeur cons. Signalons en dernier lieu la combinaison fonctionnelle, l'opérateur *, l'extension des primitives à un nombre quelconque d'arguments proposés dans [FRIEDMAN 78].

	Langage purement fonctionnel	Structure des données de base	Contrôle statique du type	Définition des types	Combinateurs	Correspondance entre les paramètres d'appels et le corps de la fonction	Caractère strict des fonctions définies	Fonctions d'ordre supérieur	Particularités
LISP [McCARTHY 60]	oui	paire pointée.	non : langage non typé.	non	non	variables.	oui	oui	- le programme LISP est lui-même une paire pointée. - ce langage fut inspiré du lambda-calcul de Church.
APL [IVERSON 62]	non : affectation et branchement.	tableau	non : langage non typé.	non	oui : ils sont nombreux mais ne s'appliquent qu'à des fonctions primitives.	variables.	oui	non	- extensions des fonctions monadiques et dyadiques connues à des arguments tableaux. - c'est le seul langage parmi ceux-ci qui ait connu un certain développement commercial.
ISWIM [LANDIN 66]	non : ordre de terminaison d'une expression (branchement).	pas de structure a priori : famille de langages.	non : langage non typé.	non	non	variables.	oui	oui	- il s'agit plus d'une famille de langages que d'un langage : on peut le spécialiser par l'ajout de structures de données et primitives orientées vers un type de problème.
[BURGE 76]	oui : bien que les ajouts de branchements et affectations soient envisagés.	paires, structures et listes.	non : langage non typé.	non	non	variables.	oui, bien que l'autre possibilité soit envisagée par Burge.	oui	- langage basé sur le lambda-calcul et constitué en lui ajoutant des facilités d'écriture (where, conditionnelle, ...) ; il est très inspiré de ISWIM.
PROLOG [ROUSSEL 76]	non : il n'est pas basé sur la notion de fonction mais sur le calcul des prédicats du 1er ordre.	liste de termes.	non : langage non typé.	non	non	pattern-matching (si on considère les sous-programmes comme des fonctions).	non : l'appel d'un sous-programme peut se terminer alors que certains arguments restent inconnus.	non	- ce langage, adapté au traitement des problème d'intelligence artificielle, possède la notion de non-déterminisme. - la version référencée (non pure) permet des effets de bord par la possibilité d'ajouts dynamiques de clauses entre autres.
FP [BACKUS 78]	oui	séquence.	non : langage non typé.	non	oui	Fonctions à un seul argument, la séquence, qui peut être décomposée par des primitives de sélection.	oui	non	- pas de notion de variable : le langage manipule uniquement des variables. - existence d'une puissante algèbre de programmes.
ML(LCF) [GORDON 79]	non : possibilité d'affectation.	pas de structure de prédilection : on peut définir des théories orientées vers une structure particulière.	oui : existence de types polymorphiques, de types abstraits ; contrôle statique du type et génération automatique s'il n'est pas spécifié.	- constructeurs de types, - types abstraits.	non	pattern-matching.	oui	oui	- ML est un langage général à l'intérieur duquel on peut en construire d'autres (PPLAMBDA, FP, ...) à travers les types abstraits et programmer des preuves sur leurs programmes. - possibilité de gérer et de générer des cas d'exceptions.
POPLAR [MORRIS 80]	oui	chaîne, liste.	non	non	oui	variables.	non	non	- possibilité d'annoter un programme par des exemples automatiquement contrôlés. - l'application d'un modèle à une chaîne permet d'en extraire une partie par "pattern-matching".
ALFA [HASERMANN 80]	oui	séquence.	oui : le domaine des fonctions doit être spécifié à leur définition. Les types peuvent être polymorphiques.	constructeurs de types.	oui	Fonctions à un seul argument, la séquence, qui peut être décomposée par des primitives de sélection.	oui	oui, mais elles n'ont pas le même statut que les fonctions (règles de construction et d'application différentes).	- ce langage rappelle beaucoup FP, mais il est d'ordre supérieur et typé. - notion de module avec déclarations publiques et privées (de types et de fonctions).
HOPE [BURSTALL 80]	oui	la liste est fournie en standard mais on peut définir des types abstraits avec leurs opérateurs (liste par exemple).	oui : le domaine des fonctions doit être spécifié à leur définition. Les types peuvent être polymorphiques.	types abstraits.	non	pattern-matching.	non, on peut introduire explicitement des constructeurs de listes paresseux.	oui	- le contrôle statique de type vérifie également que la fonction est définie de façon complète sur son domaine. - l'abstraction des types est réalisée à l'aide de modules comportant des déclarations publiques et privées.
SASL [TURNER 81]	oui	liste.	non : langage non typé.	non	pas vraiment : ce sont en fait des fonctions primitives d'ordre supérieur.	pattern-matching.	non	oui	- tous les objets (entiers, booléens, caractères, listes, fonctions) ont les mêmes droits (entrée ou résultat de fonction, composant de liste, ...). - possibilité de définir les fonctions à plusieurs arguments de façon curifiée (fonction unaire rendant une autre fonction unaire).
	oui : variables à		oui : les variables		pas vraiment : l'ex-		non : l'appel est		

4ème Partie: MISE EN OEUVRE DES LANGAGES FONCTIONNELS.

4ème Partie: Mise en oeuvre des langages fonctionnels.

1) Gestion de la mémoire.

1.1) Récupération dynamique de mémoire.

1.2) Minimisation de la consommation de mémoire.

1.3) Optimisation par transformation de programmes.

2) Schémas d'exécution.

2.1) Etude comparative des différents schémas.

2.1.1) Appel par valeur ("leftmost innermost")

2.1.2) Appel par nom ("leftmost outermost")

2.1.3) Appel par nécessité ("delay-rule" ou "call by need")

2.1.4) Appel par valeur parallèle ("parallel innermost")

2.1.5) Appel par nom parallèle ("parallel outermost")

2.1.6) Substitution totale ("full substitution")

2.1.7) Evaluation paresseuse ("lazy-evaluation")

2.2) Solutions hybrides.

3) Elimination des calculs redondants.

4) Interprétation parallèle.

4.1) Parallélisme implicite des langages fonctionnels.

L'accent a été mis jusqu'ici sur le fait que les langages fonctionnels permettent de programmer en se dégageant au maximum des contraintes de mise en oeuvre. Les avantages obtenus au niveau de l'écriture et de la manipulation des programmes ont été largement développés mais il paraît légitime de se demander maintenant s'ils ne sont pas acquis au détriment de l'efficacité: puisque l'utilisateur ignore délibérément les contraintes physiques intervenant lors de l'exécution de son programme c'est l'interpréteur qui devra réaliser les optimisations nécessaires. Pour cela des techniques nouvelles doivent être mises en oeuvre; elles concernent principalement les gestions de la mémoire et du flot de contrôle. Dans les systèmes conventionnels le programmeur indique explicitement les zones mémoire qu'il manipule (réservations) et leur durée de vie (déclarations locales, globales); il peut réutiliser une zone qui ne contient plus d'information valide (affectation); il précise exactement l'ordre des opérations à effectuer (LOOP, GOTO,...); à l'inverse dans les langages fonctionnels les structures de données sont créées dynamiquement, on ne peut les réallouer explicitement et le flot de contrôle n'est pas spécifié; la mise en oeuvre choisie doit donc permettre une exécution rapide en tenant compte des ressources disponibles et assurer la cohérence avec la sémantique attribuée au langage. Les diverses techniques proposées à cet effet ont conduit à la conception de nouvelles architectures, très éloignées des machines actuelles et plus adaptées à l'exécution des langages fonctionnels. Nous étudions la façon dont les problèmes d'allocation de mémoire sont généralement résolus avant d'examiner les principales propositions concernant les structures de machines.

1) GESTION DE LA MEMOIRE.

L'exécution d'une primitive comme "CONS" en LISP a pour effet de réserver une zone mémoire et de lui attribuer comme valeur la paire constituée de ses deux arguments; il se peut que quelque temps plus tard, ces deux champs ayant été utilisés par d'autres fonctions (CAR et CDR par exemple) la zone en question n'ait plus aucun lien avec le calcul en cours: elle doit alors pouvoir être réallouée lors d'un autre appel de CONS, en quelque sorte s'il y a allocation dynamique et automatique d'emplacements, il doit aussi y avoir récupération dynamique et automatique; faute de quoi les programmes risquent d'être bloqués par manque d'espace alors qu'une grande partie de la mémoire est réellement inutilisée. Quelques algorithmes de récupération sont désormais bien connus, voici leur différentes caractéristiques.

1.1) RECUPERATION DYNAMIQUE DE MEMOIRE

Les principaux critères lors de la mise en oeuvre d'un récupérateur dynamique sont les suivants:

- il doit pénaliser au minimum le calcul principal.
- il est souhaitable qu'il opère de façon continue de façon à ne pas entraîner de grandes variations dans les temps d'exécution des primitives.

- il est intéressant qu'il puisse être distribué dans le cas de mise en oeuvre sur multi-processeur.

- il doit consommer le minimum de place lui-même.

- il doit pouvoir être prouvé.

Examinons maintenant les caractéristiques de quelques techniques parmi les plus utilisées.

1.1.1) Compteurs de références.

Le principe est le suivant ([COLLINS 60]): le nombre de références courant à une cellule est comptabilisé dans un champ appelé "compteur": quand il devient nul la cellule est rendue à l'espace de mémoire libre. Pour des raisons d'économie ce champ est généralement assez étroit et si le compteur atteint la valeur maximale ("l'infini"), il n'en change plus; il n'augmente pas si la cellule est référencée (pour cause) mais n'est pas décrémenté dans le cas inverse: cette cellule ne sera donc jamais réallouée. L'algorithme peut être entièrement décentralisé et il s'exécute de façon continue, les actions étant effectuées à chaque fois qu'un lien vers une cellule est créé ou abandonné. Chaque utilisation de données entraine donc un calcul qui consiste en une suite d'incréments et de décréments. Un autre problème est de trouver la taille adéquate du champ compteur: trop elevée elle implique une consommation de mémoire inacceptable, trop faible elle limite la portée de l'algorithme en interdisant la réallocation d'un grand nombre de cellules. Soulignons enfin une restriction majeure pour ce genre de méthode: les structures circulaires ne peuvent être récupérées simplement puisque toute cellule

est toujours référencée au moins une fois. La preuve de l'algorithme ne pose pas de problème, le seul point à vérifier dans le cas distribué étant qu'un incrément doit être pris en compte avant un décrément fatal (conduisant à la désallocation).

1.1.2) Ramasse-miettes par marquage/balayage.

Dans ce cas chaque cellule possède un bit de marquage; quand la mémoire libre devient insuffisante les cellules utiles sont d'abord marquées puis un balayage permet de rendre disponibles celles qui n'ont pas été atteintes lors de la première phase et éventuellement de retasser les autres. Divers algorithmes de marquage ont été proposés ([KNUTH 75]): ils parcourent généralement les arbres qui prennent leurs racines dans les registres de l'interpréteur, soit en utilisant une pile, soit à l'aide de pointeurs inverses dans l'arbre (afin d'économiser de la place mémoire). La phase de balayage consiste en un parcours séquentiel de l'espace pour désallouer les cellules non marquées; il peut être également intéressant de retasser l'espace utile de façon à faciliter des allocations ultérieures. Cet algorithme possède des caractéristiques opposées au précédent: le calcul introduit est moins important (nul tant qu'il n'y a pas saturation de la mémoire), il consomme moins de place (seulement un bit par cellule). Cependant il est plus difficilement distribuable et l'inconvénient majeur réside dans les variations imprévisibles des temps d'exécution des primitives puisqu'elles peuvent éventuellement déclencher l'ensemble des tâches de récupération de place. On résout ce problème en permettant à l'algorithme de se dérouler en parallèle avec le programme utilisateur mais il faut s'assurer alors que les vitesses respectives des deux processus sont telles que le calcul principal ne soit jamais privé de mémoire et que ses accès aux cellules ne

soient pas perturbés par le processus de récupération; les preuves de tels programmes sont donc moins évidentes à établir ([DIJKSTRA 75]). Les deux algorithmes décrits jusqu'ici ont des qualités différentes et peuvent se combiner judicieusement, le ramasse-miettes n'intervenant que pour réallouer les cellules qui n'ont pu être traitées par le premier algorithme, c'est-à-dire celles dont compteur a atteint sa valeur maximum.

1.1.3) Ramasse-miettes par recopie.

Ce procédé diffère principalement du précédent par le fait qu'il fonctionne en une seule phase, la reconnaissance et le retassement des zones utiles étant réalisés simultnément ([BAKER 78]). Il n'y a donc pas de champ de marquage dans les cellules. Cependant il est nécessaire qu'une grande partie de la mémoire soit libre car les cellules utilisées doivent être recopiées dans des cellules disponibles. Le principe est le suivant: les pointeurs contenus dans les registres de l'interpréteur sont d'abord recopiés dans la mémoire libre. Ces pointeurs référencent des cellules de la mémoire occupée; la mémoire libre est alors parcourue séquentiellement afin d'y transférer toutes les cellules accessibles. Deux cas peuvent se présenter pour une cellule rencontrée en mémoire libre qui contient un pointeur vers une cellule en mémoire occupée:

- cette cellule contient un pointeur en mémoire occupée ou une valeur. Dans ce cas la cellule est recopiée en mémoire libre; sa nouvelle adresse remplace l'ancien pointeur dans la cellule qui la référençait; cette adresse est également placée à l'ancienne adresse de la cellule afin d'indiquer, si elle est référencée à nouveau, qu'elle a déjà été transférée et où elle l'a été.

- cette cellule contient un pointeur en mémoire libre: elle a donc été traitée et elle se trouve désormais à la nouvelle adresse indiquée par ce pointeur; il suffit donc de mettre ce nouveau pointeur à la place de l'ancien dans la cellule de mémoire libre qui la référence.

Dans le cas où la cellule en mémoire libre contient une valeur et non un pointeur il n'y a rien à modifier. L'algorithme se termine quand toutes les cellules en mémoire libre ont été parcourues; la mémoire dite "occupée" peut alors être récupérée et, les rôles étant inversés, elle devient la mémoire libre. Cette méthode peut être employée parallèlement au programme utlisateur; le travail peut également être réparti dans les différentes primitives d'accès aux données (CONS, CAR, CDR par exemple) ce qui assure un temps d'exécution constant. Au niveau de l'espace requis cette méthode est comparable à celle des compteurs de référence mais elle peut de plus traiter les structures circulaires. Notons toutefois que cette technique n'est pas distribuable comme la première et qu'elle nécessite un espace d'adressage beaucoup plus large.

1.2) MINIMISATION DE LA CONSOMMATION MEMOIRE.

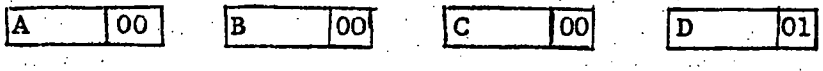
Les algorithmes de récupération dynamique étant toujours assez coûteux, ils ne doivent pas être considérés comme les seuls outils de gestion de la mémoire. Il est souhaitable au contraire de leur adjoindre des techniques permettant de réduire les besoins en mémoire du programme utilisateur: le meilleur moyen de diminuer le coût de la récupération dynamique d'espace n'est-il pas de limiter l'allocation elle-même?

Considérons d'abord la représentation des structures de données en

mémoire; le codage traditionnel des listes en LISP est réalisé à l'aide d'un ensemble de couples de cellules pointant sur les parties CAR et CDR. La liste (A.(B.(C.(D.NIL)))) est implantée en mémoire par:

et occupe donc huit cellules alors que cinq pourraient suffire.

D'autres types de représentation ont été proposés comme le "codage du CDR" ([CLARK 77]) qui consiste à diviser chaque cellule en deux parties: un champ valeur qui correspond généralement au CAR de l'expression et un couple de bits indiquant la position du CDR: s'il vaut 00 par exemple le CDR se trouve dans la cellule suivante, s'il vaut 01 le CDR vaut NIL. La liste précédente se code donc:



Cette représentation implique en moyenne une diminution de 50% de la mémoire requise car la plupart des références correspond au schéma linéaire cité ici. D'autres techniques sont également apparues comme la manipulation des pointeurs sous forme d'indices relatifs plus compacts que des adresses complètes. Le problème est cependant d'adapter la technique de récupération à la représentation choisie. La méthode des compteurs de référence par exemple s'adapte difficilement au "codage du CDR" puisque la contiguité des cellules est significative; il faut donc allouer des blocs de tailles variables ce qui est difficile dans ce cas puisqu'il n'y a pas de compactage car les zones disponibles ne sont pas forcément de taille suffisante pour accueillir un ensemble de cellules voisines.

La mémoire totale occupée par les programmes dépend également de la façon dont ils sont représentés en machine. La place nécessaire est généralement moins importante s'ils sont compilés au lieu d'être interprétés sous leur forme initiale (bien que ce procédé soit plus naturel pour les langages fonctionnels). Il est également possible de traduire automatiquement les programmes sous forme de combinateurs en faisant disparaître toutes les variables ([TURNER 79]). Le code obtenu est assez volumineux mais il n'y a plus dans ce cas de problème de gestion de l'environnement; dans les interpréteurs classiques ([LANDIN 64]) les liens entre les identificateurs et leurs valeurs sont maintenus à l'aide de listes de paires qui occupent une place importante en mémoire; ces liaisons étant le plus souvent définies de manière statique (sauf dans le LISP d'origine ([MACCARTHY 60]) les environnements de déclaration des fonctions doivent être maintenus avec leur définition. De même dans le cas d'un schéma d'évaluation paresseux ("lazy evaluation"), les arguments d'une fonction sont calculés quand ils deviennent nécessaires et leur environnement de calcul doit donc être conservé jusqu'à ce moment. Ces environnements sont généralement coûteux en place mémoire et doivent fréquemment être sauvegardés, ce qui rend nécessaire des techniques d'optimisations: par exemple ne maintenir que les parties qui sont effectivement réutilisées ([SUSSMAN 81]), ou éviter de garder plusieurs valeurs successives du même identificateur dans l'environnement ([WISE 81]).

Une autre amélioration est obtenue dans le cas de l'évaluation paresseuse pour des programmes manipulant des structures de données (listes par exemple) très importantes; en effet selon cette méthode les éléments de la liste ne sont calculés qu'au moment où ils sont demandés et peuvent disparaître dès qu'ils ont été consommés: la structure de données ne doit donc pas forcément être présente entièrement en mémoire à un instant donné et l'espace utilisé par le calcul

n'est pas nécessairement proportionnel à la taille des structures manipulées ([MORRIS 80], [WISE 81]).

1.3) OPTIMISATION PAR TRANSFORMATION DE PROGRAMMES.

Les divers procédés de transformation ayant déjà été étudiés dans la deuxième partie (paragraphe 3) nous allons plutôt examiner ici leur utilisation pour réduire l'espace mémoire nécessaire à l'exécution. Le système de Burstall et Darlington décrit dans [DARLINGTON 76] a pour but de produire, à partir d'une version fonctionnelle, un programme séquentiel le plus proche possible de ce qu'aurait écrit directement un "bon" programmeur dans un langage conventionnel. En particulier une cellule mémoire devenue inutile sera réutilisée si une demande d'espace intervient ultérieurement (par un CONS par exemple). Le système procède en calculant des états symboliques et en réalisant une sorte de ramasse-miettes statique, les primitives d'allocation de mémoire étant éventuellement remplacées par réaffectation d'une mémoire devenue inutile; considérons l'exemple suivant qui décrit un programme d'inversion de liste (il n'est pas directement sous forme fonctionnelle car il a déjà subi certaines transformations):

Exemple 1.

```
result:= nil;
while not null(X) do
 begin
  result:= cons(hd(X),result);
  X:= tl(X)
 end
```

Il utilise une cellule supplémentaire par appel récursif et en consomme donc un nombre égal à la longueur de la liste en argument. Si val(identificateur) représente la valeur (pointeur ou valeur immédiate) d'un identificateur et si hd(pointeur) et tl(pointeur) délivrent les valeurs des deux champs de la cellule indiquée par "pointeur" l'état symbolique en début d'une itération peut être décrit par:

```
     val(X)=C1; hd(C1)=a1; tl(C1)=C2;
                   hd(C2)=v3; tl(C2)=v4;
val(result)=C3; hd(C3)=v1; tl(C3)=v2;
```

Après les deux affectations, il devient:

```
     val(X)=C2; hd(C2)=v3; tl(C2)=v4;
val(result)=C4; hd(C4)=a1; tl(C4)=C3;
                   hd(C3)=v1; tl(C3)=C2;
                   hd(C1)=a1; tl(C1)=C2;
```

Le ramasse-miettes indique que C1 est désormais inutile et pourrait donc être utilisée au lieu de réquisitionner C4. Le système produit alors les affectations qui permettent une telle transformation (éventuellement en ajoutant une variable intermédiaire) et le programme suivant est obtenu automatiquement:

```
result:= nil;
while not null(X) do
begin
   newvar:= tl(X);
   tl(X):= result;
```

```
    result:= X;
    X:= newvar;
end
```

Ce programme n'entraîne pas l'allocation dynamique de mémoire ce qui soulagera d'autant la récupération au moment de l'exécution.

Une autre optimisation fréquemment évoquée est le remplacement des formes récursives en formes itératives ([DARLINGTON 76], [BURSTALL 77]) qui évitent les stockages dans les piles. Wadler ([WADLER 81]) quant à lui propose un système entièrement automatique permettant entre autres de supprimer la création de listes intermédiaires pendant le calcul. Pour ce faire, il associe une transformation à chaque combinaison d'opérateurs. Par exemple dans une syntaxe FP:

$$(\alpha f) \circ (\alpha g) \rightarrow \alpha (f \circ g)$$

On voit que la nouvelle forme réalise l'économie d'une séquence en effectuant les deux tâches en un seul parcours.

2) SCHEMAS D'EXECUTION.

Puisque dans les langages fonctionnels l'ordre des opérations à effectuer n'est pas indiqué explicitement par le programmeur, il s'agit de le déterminer automatiquement au niveau de l'interprétation. Le choix qui est alors fait a des conséquences au niveau de l'efficacité de la mise en oeuvre, le chemin pour parvenir à la solution pouvant être plus ou moins long, mais aussi au niveau de la sémantique du langage: certaines exécutions peuvent se terminer dans un cas et boucler indéfiniment dans l'autre.

Exemple 2.

F(x,y)=si x=0 alors 0 sinon F(x+1,F(x,y))*F(x-1,F(x,y))

1 2 3 4

si on choisit d'évaluer d'abord les appels récursifs les plus imbriqués (stratégie "parallel innermost", c'est à dire les occurences 2 et 4 de F) le programme ne se termine pas puisque F(x,y) conduit à calculer F(x;y); si on évalue d'abord les appels les plus externes ("parallel outermost", c'est à dire les occurences 1 et 3 de F) on arrive au résultat 0 (la primitive * étant considérée non stricte c'est-à-dire $\perp * 0 = 0 * \perp = 0$) tandis qu'en calculant seulement l'appel le plus externe à droite (c'est à dire l'occurence 3 de F) on obtient la même valeur en réalisant moins de travail (si les arguments sont des entiers positifs). Ce critère d'efficacité n'est évidemment pas général et il est variable selon les programmes proposés et les ressources physiques disponibles (processeurs, mémoires).

Détaillons d'abord les avantages respectifs des différents schémas avant de

voir comment ils sont mis en oeuvre.

2.1) ETUDE COMPARATIVE DES DIFFERENTS SCHEMAS

Suivant la description de Cadiou ([CADIOU 72]) et Vuillemin ([VUILLEMIN 74]), considérons l'exécution d'un programme comme un cycle composé de deux opérations: la simplification et la substitution.

- la simplification (standard) correspond grossièrement à l'exécution de primitives du langage à condition qu'elles disposent des arguments suffisants:
  if T then A else F(F(X)) par exemple se simplifie en A.
- la substitution consiste en le remplacement d'un appel de fonction par sa définition correctement unifiée, et peut donc être assimilée à l'exécution d'une fonction utilisateur.

Exemple 3.
Si f est définie par:

f(x) = si x = 0 alors 0 sinon f(x-1)

l'expression: f(x+1) + f(1) peut se transformer en:

(si x+1 = 0 alors 0 sinon f((x+1)-1)) + f(1)

par substitution de l'appel le plus à gauche.

La simplification est donc uniquement concernée par la définition des primitives du langage tandis que les schémas d'exécution qui nous intéressent ici sont des ordres sur les différentes substitutions possibles. Il en existe un grand nombre en théorie mais nous considérons seulement les plus intéressants.

2.1.1) Appel par valeur ("leftmost innermost").

La substitution réalisée à chaque étape est celle de la fonction la plus imbriquée la plus à gauche. Cette méthode est utilisée de façon classique ([MACCARTHY 60], [LANDIN 64]) car c'est la plus simple à mettre en oeuvre: elle revient à calculer séquentiellement les arguments d'une fonction avant l'appel. La principale faiblesse apparaît dans le cas où la fonction n'a pas besoin de tous ses arguments pour rendre son résultat car les arguments inutiles sont évalués et peuvent même, dans le pire des cas, ne pas se terminer; il faut donc vérifier que la sémantique du langage s'accomode de cette restriction. Ainsi dans l'exemple 2 l'exécution de F(x,y) conduit à celle de F(x,y) (pour $x \neq 0$) et ainsi de suite. Un autre inconvénient déjà cité (section 3.2.2 de la deuxième partie) réside dans le fait que cette méthode invalide certains schémas de transformation de programmes comme le pliage/dépliage.

2.1.2) Appel par nom ("leftmost outermost").

Cette fois c'est la fonction la plus externe la plus à gauche qui est lancée d'abord. Les programmes implémentés de cette manière sont plus définis que par l'appel par valeur. Par exemple si f est definie par:

f(x) = si x = 0 alors 0 sinon f(x-1,f(x))

f(x) = 0 pour x>0 alors qu'avec la méthode précédente f(x) n'est définie que pour x=0. Cependant la fonction F de l'exemple 2 reste indéfinie pour tout (x,y) tel que x>0. Ceci vient du fait que la fonction * doit être stricte et symétrique: elle nécessite donc une évaluation de ses arguments en parallèle et non pas séquentiellement comme ici. Ainsi, alors que la méthode précédente entraînait le calcul de tous les arguments d'une fonction une et une seule fois, l'appel par nom permet d'éviter le calcul de certains arguments mais elle provoque aussi le recalcul de ceux qui sont utilisés plusieurs fois par la

fonction comme le montre l'exemple suivant:

Exemple 4.

f(x) = si x>0 alors x-1 sinon f(f(x+2))

f(0) conduit à f(f(2)) soit:

si f(2) > 0 alors f(2)-1 sinon f(f(f(2)+2))

et l'argument f(2) est calculé deux fois.

De plus la mise en oeuvre de cette stratégie est plus coûteuse puisque les arguments d'une fonction doivent être maintenus sous forme d'expression non calculée avec leur environnement afin de permettre un calcul ultérieur ([BURGE 76], [WISE 81]). Notons que cette restriction est écartée dans la mise en oeuvre de Turner ([TURNER 79]) qui utilise cet ordre d'évaluation de façon naturelle après avoir supprimé les variables du programme. Cette méthode invalide de plus les techniques standard d'élimination de la récursivité conduisant à une "récursivité à droite"; ainsi la fonction suivante:

f(x,y) = si x = 0 alors y sinon f(x-1,y+1)

n'est pas sous forme itérative puisque les additions y+1 ne sont effectuées que rétroactivement quand x=0 est vrai: la sauvegarde des expressions intermédiaires est donc nécessaire.

2.1.3) Appel par nécessité ("delay-rule" ou "call by need").

Cette technique est basée sur l'appel par nom: les conditions de terminaison sont identiques mais la méthode est optimisée car les arguments sont évalués au maximum une fois dans ce cas. Cette stratégie est optimale au sens du nombre de calculs effectués dans le cas d'un langage séquentiel ([VUILLEMIN 74]), c'est à dire où l'on peut déterminer à tout moment le calcul nécessaire

sans avoir à en exécuter d'autres ([HUET 79]). L'appel par nécessité est généralement mis en oeuvre en partageant l'accès aux arguments (pointeurs) au lieu d'en manipuler des copies ([TURNER 79]).

2.1.4) Appel par valeur parallèle ("parallel innermost").

Contrairement à l'appel par valeur où les arguments sont évalués séquentiellement, ils sont calculés ici de façon simultanée. La définition des fonctions est la même, la seule différence étant que cette stratégie est plus adaptée à une interprétation parallèle du langage. Les mise en oeuvres dirigées par les données ("data flow") ([DENNIS 79], [MACGRAW 82]) et certaines machines à réduction ([MAGO 80], [TRELEAVEN 80]) fonctionnent selon ce principe.

2.1.5) Appel par nom parallèle ("parallel outermost").

Cette fois toutes les fonctions les plus externes sont substituées en parallèle. Cette stratégie implique une plus grande définition des fonctions que ne le permet l'appel par nom comme en témoigne l'exemple 2. De plus elle permet comme la précédente, de mettre à profit le parallélisme intrinsèque contenu dans les langages fonctionnels. Notons toutefois une nouvelle difficulté de mise en oeuvre: il faut s'assurer que les processus calculant des arguments devenus inutiles ne perturbent pas le système et que le résultat final pourra être rendu même s'ils sont enlisés dans une boucle infinie([GRIT 81], [BAKER 77]).

2.1.6) Substitution totale ("full subsitution").

Il s'agit du cas extrême où toutes les substitutions pouvant avoir lieu sont menées en parallèle. Cette technique implique une définition maximum pour les fonctions mais les problèmes posés à la mise en oeuvre sont importants (plus

la définition est large plus la réalisation est compliquée en général); il faut ajouter à ceux cités pour l'appel par nom et l'appel par nom parallèle la multiplication des processus requis qui entraîne un coût de gestion important et n'est pas toujours supportable par l'architecture sous-jacente. C'est pourquoi cette stratégie n'est jamais utilisée sans amémagements permettant de réduire cette prolifération exponentielle, par exemple en retardant le déclanchement des processus qui ne sont pas sûrs de produire un résultat nécessaire au calcul principal ([KELLER 79], [SLEEP 81a]) ou en récupérant ceux qui sont engagés dans un calcul inutile ([GRIT 81], [BAKER 77]).

2.1.7) L'évaluation paresseuse ("lazy evaluation").

Le principe est celui de l'appel par nécessité mais il est appliqué de manière plus fine dans le sens où les structures de données peuvent être calculées partiellement, à la demande. De cette manière si seul le premier élément d'une liste argument est nécessaire au calcul de la fonction, le reste de la liste ne sera pas évalué ([FRIEDMAN 76], [HENDERSON 76]); c'est ce qui se passe dans l'exemple 5:

Exemple 5.

car[car[cons[cons[f[x];g[x]];h[x]]]]

g[x] et h[x] resteront "suspendus" puisqu'ils ne sont pas accédés directement par une primitive car ou cdr.

On se ramène donc à l'appel par nécessité en considérant les listes comme une suite d'arguments plutôt qu'un argument composé. Cette technique peut être implémentée en LISP en considérant que la primitive CONS est non stricte en ses deux arguments et rend deux suspensions constituées d'une expression et de son

environnement. Ces suspensions ne sont réveillées (ou exécutées) que lorsqu'u CAR ou CDR est effectivement appliqué à la structure. En utilisant ce primitives dans le corps même de l'interpréteur classique ([FRIEDMAN 76]) o généralise la "lazy evaluation" à tous les appels de fonctions puisque le environnements sont crées de façon paresseuse par CONS. Henderson ([HENDERSO 76]) propose un autre interpréteur agissant par modifications successive d'états de mémoire, le résultat final prenant la place de l'expression à calculer. Dans ce modèle l'exécution de CONS ne produit aucune modification de la mémoire, mais ce sont les calculs de CAR et CDR qui déclenchent l'évaluation des cellules référencées. De plus il est montré que la valeur obtenue en résultat ne dépend pas de l'ordre d'évaluation des différentes expressions (propriété de Church-Rosser) ce qui valide la méthode. Le principal inconvénient de cette stratégie est le grand nombre de sauvegardes d'environnements (à travers les suspensions) qu'il est nécessaire de réaliser ce qui entraîne à la fois des pertes de temps et d'espace mémoire. Une conséquence intéressante est par contre la possibilité de manipuler des listes potentiellement infinies, à condition bien entendu de ne pas tenter de les parcourir entièrement. Nous ne revenons pas sur l'économie de mémoire qu'il est possible de réaliser en n'imposant pas aux listes d'être entièrement présentes à un moment donné (section 1.2)).

2.2) SOLUTIONS HYBRIDES.

Ces diverses stratégies possèdent, comme nous l'avons constaté, des qualités très différentes, elles sont parfois utilisées de manière moins systématique et combinées entre elles afin de profiter des points positifs de chacune dans des cas précis. Nous avons déjà évoqué la stratégie utilisée par

Keller ([KELLER 79]) et Sleep ([SLEEP 81b]). Elle est basée sur l'évaluation paresseuse mais le parallélisme est introduit en permettant le calcul simultané des arguments des primitives strictes (comme + par exemple); on est ainsi assuré de n'effectuer aucun calcul inutile tout en atteignant un degré de parallélisme intéressant. Notons toutefois que seules les primitives strictes conduisent à un calcul en parallèle des arguments, les autres fonctions, même si elles le sont aussi (il faudrait pouvoir le déterminer), seront évaluées de façon paresseuse. Une solution proposée parfois pour résoudre cette question est de permettre à l'utilisateur d'indiquer à quel moment il désire entamer l'évaluation des expressions ([PRINI 80]) mais de tels procédés vont à l'encontre des principes mêmes de la programmation fonctionnelle. Plus prometteuse semble la méthode de Mycroft ([MYCROFT 80]) qui permet de repérer automatiquement et de façon statique un grand nombre de cas où le remplacement de l'appel par nécessité par l'appel par valeur est correct, c'est-à-dire qu'il ne peut entraîner de modification du résultat final. Un argument peut être évalué à l'avance dans deux situations:

- cet argument est toujours utilisé dans le corps de la fonction: donc s'il est indéfini l'appel de la fonction à laquelle il est transmis doit l'être aussi.
- cet argument ne peut diverger; au pire il sera donc calculé inutilement mais il ne perturbera pas le résultat final.

Pour cela deux prédicats sont associés à chaque fonction f; ils ont le même nombre de paramètres que f et ceux-ci sont des valeurs booléennes qui symbolisent la convergence des arguments de f (on dit qu'un argument diverge si son évaluation délivre un résultat indéfini et converge sinon); ces prédicats sont:

- f# qui indique une condition nécessaire de divergence de la fonction:
  si f#(b1,...,bi-1,faux,...,bn) = faux, alors gi peut être transmis par valeur lors de l'appel f(g1,...,gn) car sa divergence implique celle de l'appel de f.

- fb qui indique une condition nécessaire de convergence de la fonction:
  si gib(x1,...,xk) = vrai ∀ x1,...,xk, alors gi peut être transmis par valeur lors de l'appel f(g1,...gn) car il ne peut diverger.

Par exemple si IF est la conditionnelle

IF# (p,x,y) = (p et (x ou y))

ce qui signifie que IF utilise obligatoirement p et l'un de ses deux autres arguments (x ou y).

IFb (p,x,y) = (p et (x et y))

ce qui signifie que pour que IF converge à coup sûr il faut que ses trois arguments convergent.

f# permet de trouver les arguments qui sont forcément utilisés par la fonction tandis que fb sert à repérer ceux qui ne peuvent diverger. Ainsi dans le cas

F(x,y) = G(x,y+1)+y

on sait que +# (x,y) = (x et y)

et +b (x,y) = (x et y)

donc F# (b,faux) = (G#(b,vrai) et faux) = faux et y peut être passé par valeur; donc y dans le corps de F est déjà calculé; si on note E# et Eb les fonctions similaires applicables à une expression on peut écrire Eb(y) = vrai donc Eb(y+1) = vrai ce qui signifie que y+1 converge forcément et qu'il peut donc être passé à la fonction par valeur (c'est-à-dire calculé avant l'appel).

On peut donc conclure à que le deuxième argument des deux fonctions F et G peuvent ici être calculés sans dommages avant l'appel des fonctions, le premier parce qu'il est utilisé dans tous les cas et le second car il ne peut diverger. Cette technique n'est cependant pas générale; par exemple le cas

F(x,y) = si P(x,y) alors y sinon 0

avec P(x,y) toujours vrai, n'est pas reconnaissable en général. De plus elle devrait être raffinée pour traiter en particulier l'évaluation paresseuse.

Les qualités d'un schéma d'exécution, la correction et l'optimalité, ont été étudiées dans deux cadres différents: les schémas de programmes récursifs et le lambda-calcul. On sait ([VUILLEMIN 74]), que la substitution totale et l'appel par nom parallèle sont toujours corrects tandisque l'appel par nom ne l'est que pour un langage séquentiel. Ces résultats ont été étendus au lambda-calcul par Levy ([LEVY 78]). On peut montrer qu'un schéma est correct d'une manière purement syntaxique ([DOWNEY 76]) en vérifiant que tout terme obtenu par la règle "parallel-outermost" peut être dérivé en un terme obtenu par ce schéma. Si l'optimalité de l'appel par nécessité a été établie dans le cas séquentiel ([VUILLEMIN 74]), le problème est moins simple pour un langage parallèle. Cependant Berry et Levy ([BERRY 79]) ont montré l'existence de règles optimales pour une classe d'interprétations englobant les interprétations séquentielles. Le formalisme de Huet et Levy ([HUET 79]) permet de définir les notions de systèmes séquentiels (propriété indécidable) et fortement séquentiels (décidable) pour lesquels existe un interpréteur très efficace. La question est donc bien cernée théoriquement mais des progrès restent à réaliser au niveau de la mise en oeuvre, notamment sur l'efficacité relative de ces différents schémas dans une configuration donnée.

3) ELIMINATION DES CALCULS REDONDANTS.

L'exemple le plus célèbre de programme conduisant, dans sa forme la plus immédiate, à une importante quantité de calculs redondants est celui de la fonction de Fibonacci. Il s'écrit de façon naturelle:

Exemple 6.

```
fib(x) = si x <= 1 alors 1
         sinon fib(x-2) + fib(x-1)
```

ce qui implique par exemple pour fib(4) que fib(2) est calculé deux fois et fib(1) trois fois.

Une solution naturelle consiste à stocker les résultats des appels de fonctions qui peuvent être renouvelés (technique de "memo"). Le principal problème est alors de savoir où s'arrêter car il n'est pas question de garder en mémoire tous les résultats des appels de fonctions avec les arguments associés. Les méthodes automatiques étant assez limitées, il est alors proposé à l'utilisateur d'indiquer quels appels doivent être mémorisés; cette solution est inélégante et convient peu à un langage fonctionnel. D'autre part le problème de la durée de vie des valeurs ainsi maintenues reste posé et dans le cas distribué celui de la compétition pour l'accès à la zone de sauvegarde.

Une autre proposition ([PETTOROSSI 80]) consiste à introduire explicitement des communications entre les différents sous-calculs d'un programme afin que chacun puisse faire connaître ses résultats aux autres; là encore on sort du

cadre fonctionnel.

Comme nous l'avons vu dans la seconde partie (section 3.3), les transformations de programmes se révèlent à nouveau être une solution agréable dans ce cas puisque la méthode pliage/dépliage de Burstall et Darlington ([BURSTALL 77]) permet d'obtenir la nouvelle définition:

f(x) = si x <= 2 alors 1 sinon
u+v où <u,v> = g(x)

et g(x) = si x = 0 alors <1,1> sinon
<u+v,u> où <u,v> = g(x-1)

et les calculs redondants sont donc éliminés.

4) INTERPRETATION PARALLELE.

A l'heure où la technologie permet d'envisager la connection d'un nombre toujours plus élevé de processeurs, il paraît intéressant d'étudier de plus près de quelle manière les langages applicatifs peuvent s'adapter à une telle situation, c'est-à-dire comment les programmes fonctionnels peuvent se décomposer en un ensemble de tâches exécutables en parallèle. On peut en fait distinguer deux problèmes dans l'interprétation parallèle: il faut d'abord exhiber le parallélisme contenu dans une expression fonctionnelle puis implanter la solution obtenue sur l'architecture parallèle réellement disponible de façon à optimiser l'utilisation des ressources et à minimiser le surplus de coût de gestion introduit.

4.1) PARALLELISME IMPLICITE DES LANGAGES FONCTIONNELS.

Il existe principalement deux sources de parallélisme dans les langages fonctionnels:

- l'évaluation simultanée de tous les arguments d'une fonction,

- le déclanchement du calcul de la fonction en parallèle avec celui de ses arguments.

La facilité avec laquelle le parallélisme inhérent aux langages fonctionnels peut être exhibé s'explique tout d'abord par l'absence de notions d'affectation, de variables partagées; deux expressions différentes sont

toujours exécutables en parallèle sans précaution particulière, la seule synchronisation nécessaire étant celle qui est impliquée par la dépendance logique du calcul; ainsi les arguments d'une fonction peuvent être évalués simultanément. L'exécution de la fonction peut aussi débuter en même temps mais elle est bloquée si elle accède un argument non évalué. Il en va autrement dans les langages conventionnels car les expressions peuvent produire des effets de bord (modification d'une variable extérieure à l'expression) et l'ordre dans lequel elles sont évaluées n'est donc pas indifférent. Le fait que le flot de contrôle ne soit pas explicite permet, comme nous venons de le voir dans la section 2.1 d'interpréter de façon parallèle un programme qui n'a pas forcément été écrit dans cet esprit. Les problèmes de gestion du parallélisme sont donc complètement ignorés de l'utilisateur. La principale difficulté d'un interpréteur fonctionnel parallèle ne réside pas tant dans la création d'un parallélisme massif que dans sa limitation à un parallélisme utile. A quoi sert par exemple de lancer un processus pour calculer une fonction qui accède immédiatement à des arguments inévalués, à quoi sert de créer un nombre colossal de processus si la machine sous-jacente ne dispose pas d'un nombre de processeurs suffisant? Dans ce cas le parallélisme n'aura pour effet que de ralentir l'exécution totale car le gain de temps obtenu par la superposition des tâches sera plus que compensé par le coût de communication, de gestion du parallélisme et l'exécution éventuelle de tâches inutiles retardant le calcul principal.

4.2) ADAPTATION AUX RESSOURCES DISPONIBLES.

Le problème se situe en fait à deux niveaux: il faut engendrer un nombre de processus tenant compte, si possible, de l'état courant d'occupation des

processeurs et il faut également assurer une répartition géographique équilibrée des processus; il ne doit pas y avoir par exemple de processeurs surchargés alors que d'autres sont oisifs. Cette dernière question est directement liée au type d'architecture utilisé et elle est donc traitée dans le chapitre 5. On s'intéresse plutôt ici à la limitation du nombre de processus. La plupart des mises en oeuvre appliquent en réalité l'une des stratégies d'exécution définies plus haut de façon systématique, c'est-à-dire sans tenir compte des ressources réellement disponibles. Voici toutefois quelques propositions intéressantes pour tenter de résoudre ce problème.

4.2.1) Le ramasse-miettes des processus par marquage.

Le schéma de Baker et Hewitt ([BAKER 77]) est basé sur la substitution totale; le maximum de parallélisme est ainsi créé mais des dispositions sont prises d'une part pour réaffecter les processus occupés à des tâches devenues inutiles, d'autre part pour permettre aux processus jugés les plus urgents de s'exécuter de façon prioritaire quand il y a conflit d'accès aux processeurs. L'algorithme de récupération des processeurs a l'originalité d'être assimilé à celui de récupération de la place mémoire (ramasse-miettes par recopie). Chaque processus est représenté en mémoire par un vecteur d'état contenant les valeurs de ses registres. Le marquage commence à partir du vecteur d'état du processus principal et se poursuit normalement. Les processeurs occupés utilement verront leur vecteur d'état marqué et pourront alors continuer leur travail (en parallèle avec le récupérateur) tandis que les autres sont réaffectés puisqu'ils n'ont plus de lien avec le calcul principal. Cet algorithme, dont la validité est montrée de façon informelle présente toutefois l'inconvénient de considérer que tous les processeurs partagent le même espace d'adressage. De plus il

nécessite l'arrêt complet des processus de façon périodique, même s'ils sont réactivés dès qu'il ont été reconnus utiles.

La méthode utilisée pour régler les conflits entre processus consiste à leur accorder une importance décroissant exponentiellement avec leur profondeur dans l'arbre d'activations; chaque processus conserve 50% du potentiel d'exécution qui lui est fourni et en attribue 50% à sa descendance. Ainsi les listes potentiellement infinies peuvent être manipulées malgré la stratégie de "full substitution" puisque les éléments ne seront plus calculés à partir d'un certain rang à cause de la faiblesse de leur potentiel d'exécution. Cet algorithme ne constitue pas une méthode infaillible mais plutôt une heuristique pour l'allocation des processeurs et il doit être accompagné du récupérateur décrit plus haut.

4.2.2) La récupération des processus par compteurs de références.

Cette méthode, proposée par Grit et Page ([GRIT 81]) s'appuie également sur le schéma de "full substitution" et le procédé de récupération des processus s'inspire aussi d'une technique utilisée pour la gestion de la mémoire, à savoir celle des compteurs de références. Dans ce système les descripteurs de processus peuvent être référencés au maximum deux fois: dans la mémoire des descripteurs qui donne une image de la hiérarchie des tâches, et dans la liste de gestion qui contient l'ensemble de travaux en attente d'exécution. Deux bits du descripteur servent de compteur de référence: le bit DONE qui indique si le processus apparaît dans la liste de gestion et le bit KILLED qui signale si le processus est référencé dans l'arbre d'activation.

Lorsqu'un processeur obtient un processus, il vérifie l'état du bit KILLED: s'il est positionné (c'est-à-dire si le processus ne fait plus partie de l'arbre d'activation) il rend le descripteur en mémoire disponible et demande une autre tâche; sinon il tente d'exécuter le processus. S'il ne peut le faire parce qu'un argument n'est pas évalué il rend le processus à la liste de gestion; s'il peut le terminer le champ DONE du descripteur est positionné à 1 et le processus "tueur" est appelé sur le sous-arbre associé pour récupérer tous les processus de sa descendance qui sont désormais inutiles (il peut en rester effectivement si la fonction considérée est non stricte). Le "tueur" parcourt le sous-arbre en question: il désalloue les descripteurs ayant le champ DONE à 1 (n'étant donc pas référencés dans la liste de gestion) et se contente de positionner le champ KILLED à 1 pour les autres. Ils ne sont pas récupérés immédiatement, car encore référencés dans la liste de gestion, mais ils le seront dès que les processus correspondant seront alloués à un processeur.

Dans ce cas de la récupération des tâches inutiles se déroule en parallèle avec le programme usager; les processus "tueurs" étant plus prioritaires que les autres on peut penser qu'ils doivent fatalement finir par récupérer tous les processus inutiles mais ceci n'est pas évident à prouver. Supposons en effet une tâche inutile générant de façon récursive et infinie de nouvelles tâches; le problème est de savoir si le tueur pourra ou non stopper cette reproduction et une démonstration doit faire intervenir les vitesses relatives des deux processus.

Signalons pour conclure à propos des compétitions d'accès aux processeurs qu'un mécanisme de priorité est possible dans ce système, qui peut servir à régler ces conflits mais rien n'indique comment calculer ces priorités. Sleep

5.1) DIFFERENTES CLASSES D'ARCHITECTURES.

La classification utilisée ici est due à Treleaven ([TRELEAVEN 82]). La première distinction qui est faite entre les machines concerne le modèle opérationnel:

- dans le modèle data-flow les programmes sont généralement représentés par des graphes dirigés représentant le flot de données entre les instructions. L'exécution des instructions est conditionnée par un ensemble de signaux de données qui indiquent la présence des arguments et leur valeur. Ces données sont contenues dans le corps des instructions et les résultats sont recopiés partout où ils sont utilisés. Le contrôle est réalisé de façon parallèle et toutes les instructions qui disposent de leurs signaux peuvent s'activer simultanément.

- dans le modèle par réduction les programmes comme les données sont considérés comme des expressions. L'exécution se résume à une succession de réductions jusqu'à l'obtention d'une expression irréductible qui est le résultat. On distingue la réduction de graphe où les arguments communs à plusieurs expressions sont partagés par référence de la réduction par chaîne où les arguments communs sont dupliqués. La première méthode permet d'éviter des calculs redondants tandis que la seconde permet une exécution distribuée. Signalons encore que plusieurs expressions peuvent être réductibles à un instant donné et les réductions peuvent être effectuées simultanément ou selon un ordre particulier.

La seconde distinction que nous ferons ici a rapport avec l'organisation physique de la machine.

- Nous ne nous étendrons pas sur le modèle centralisé dont les machines Von Neumann constituent un bon exemple.

- Les machines à communication de paquets sont décomposées grossièrement en trois parties: processeurs, mémoires et moyens de communications. Chaque partie est constituée d'un groupe de ressources identiques auquel est attribué un ensemble de travaux; dès qu'une ressource achève un travail elle en extrait un nouveau de l'ensemble qui lui est proposé. Le programme est considéré comme un ensemble de paquets d'informations pouvant s'exécuter indépendamment. Ces paquets peuvent se reproduire (création de tâche) et disparaître (fin d'exécution); la machine est d'autant plus parallèle qu'elle contient un grand nombre de ressources identiques.

- Les machines à manipulation d'expressions se composent d'un grand nombre d'unités regroupant processeurs, mémoires et communications, organisées selon une structure régulière (arbre, vecteur). La proximité physique des unités est significative dans ce cas et elle doit refléter la structure du programme traité. Chaque unité possède une part de l'expression totale qu'elle cherche à exécuter; des protocoles doivent être établis pour le cas où une unité ne dispose pas d'informations suffisantes pour travailler, afin d'éviter par exemple les conflits entre deux unités cherchant à exécuter la même portion de programme.

Remarquons, en guise de conclusion, que si les machines à manipulation d'expressions et les machines par paquets semblent plus adaptées respectivement aux modèles réduction et data-flow, tous les cas de figure sont en fait possibles. On peut dire de plus que le modèle à réduction s'accomode de tous les schémas d'exécution tandis que le modèle data-flow supporte plus

mentionne également l'utilisation des priorités ([SLEEP 81b]); dans ce système les processeurs connaissent leur charge de travail et celles de leurs voisins et il est suggéré d'accorder une priorité plus élevée aux processus les plus profonds dans l'arbre d'activation si le système est surchargé (on espère ainsi limiter la création de processus parallèles) et l'inverse sinon. Le système des priorités semble attractif vu sa souplesse d'utilisation mais il introduit un travail de gestion supplémentaire et il reste à trouver un algorithme de calcul précis. Citons également dans ce domaine la proposition de [FRIEDMAN 78] qui part d'un schéma de "lazy evaluation" et souhaite introduire une source de parallélisme en activant par avance les tâches ("sergeants") ayant le plus de chance d'être demandées par le calcul principal ("colonel"): tout le problème réside dans le choix de ces tâches.

5) STRUCTURES DE MACHINES ADAPTEES AUX LANGAGES FONCTIONNELS.

L'un des griefs de Backus à l'encontre des langages conventionnels est leur trop grande parenté avec les architectures classiques Von Neumann caractérisées grossièrement par un processeur unique, une organisation mémoire linéaire de cellules de taille fixe et un contrôle séquentiel centralisé. Au contraire les langages fonctionnels proposés en alternative sont très éloignés des structures de machines actuelles; c'est pourquoi, bien qu'il existe désormais des mises en oeuvre très efficaces sur ces architectures, on ne considère plus le schéma Von Neumann comme le seul possible et les recherches sur la mise en oeuvre des langages fonctionnels rejoignent désormais celles concernant les nouvelles structures de machines. L'étude approfondie de ces nouvelles structures et leur comparaison selon les critères en usage dans le domaine du matériel sort du cadre de cette étude. Nous allons nous contenter ici d'examiner les principales structures de machines virtuelles ou réelles proposées dans le cadre d'applications fonctionnelles et leur qualités relatives en tant que support pour la mise en oeuvre de tels langages. Avant de citer quelques exemples de machine, nous faisons d'abord ressortir quelques caractéristiques générales qui vont nous permettre de les séparer en plusieurs grandes catégories.

naturellement les stratégies "innermost". Après avoir énuméré leurs principales distinctions nous allons maintenant illustrer quelques exemples d'architectures adaptées aux langages fonctionnels.

5.2) QUELQUES MACHINES

La machine data-flow du MIT ([DENNIS 75]) est un bon exemple d'architecture par paquets. Elle est composée de cinq unités:

- unité de mémoires,
- unité de processeurs,
- unité de distribution qui transfère les paquets de données des processeurs vers la mémoire,
- réseau d'arbitrage offrant aux processeurs des instructions exécutables,
- réseau de contrôle distribuant des paquets de contrôle des processeurs vers la mémoire. En effet des signaux de contrôle sont ajoutés aux signaux de données dans les instructions pour indiquer que toutes les données se trouvent sur leur port de sortie ont effectivement été transmises et que l'instruction peut donc être réactivée.

Cette machine, qui réalise le modèle "parallel-innermost", a servi à implémenter le langage VAL ([MACGRAW 82]).

ALICE ([DARLINGTON 81b]) possède aussi une structure par paquets mais elle opère par réduction de graphes en parallèle. Le programme est un graphe dont chaque noeud ou feuille est représenté par un paquet. Chaque paquet est composé de trois champs primaires et trois champs secondaires. Les premiers indiquent respectivement:

- le numéro du paquet,
- la fonction représentée par le paquet s'il s'agit d'un noeud, le type de la

donnée si c'est une feuille,

- l'adresse (numéro du paquet) de la liste d'arguments ou la valeur de la donnée dans le cas d'une feuille.

Les champs secondaires servent au contrôle; ils sont au nombre de trois:

- le compteur de références est utilisé pour la récupération dynamique de mémoire (cette méthode est possible car il n'y a pas de graphe cyclique),
- le "status" contient entre autres le nombre des arguments manquant au noeud pour qu'il puisse s'exécuter,
- la liste des signaux contient les numéros de paquets qui attendent le résultat de la réduction du noeud.

Quand un noeud ne peut s'exécuter, faute d'arguments, il mémorise le nombre de valeurs attendues, introduit son numéro dans les listes de signaux des paquets qui les calculent et se met en attente. Chaque signal décrémente le compteur du "status" et son passage à zéro, indiquant la présence de tous les arguments, provoque le réveil du noeud qui peut alors être réduit. D'autres champs du "status" indiquent si la fonction doit être évaluée en mode paresseux ("lazy evaluation") et si elle a été effectivement demandée. Dans le cas où elle l'a été, elle est candidate à une réduction mais son caractère paresseux sera propagé de sorte que le minimum de calculs soit effectué. La détermination de ce champ "lazy" peut être laissée à l'utilisateur ou être automatisée grâce à un système du type de celui de Mycroft ([MYCROFT 80]) évoqué plus haut. Cette machine a été utilisée par le langage HOPE ([BURSTALL 80]) mais peut s'accomoder d'une grande variété de langages (d'ordre supérieur, non déterministe,...), c'est donc un instrument très général, tant en ce qui concerne les shémas d'éxécutions que les langages de programmation.

ZAPP (Zero Asignment Parallel Processor) ([SLEEP 81a]) est une machine à réduction basée sur l'organisation data-flow. Les programmes sont représentés sous forme combinatoire ([TURNER 79]) et le schéma d'exécution choisi consiste à combiner parallélisme et évaluation paresseuse en exécutant simultanément les arguments d'un combinateur strict; on est ainsi certain de n'effectuer que le calcul nécessaire et la récupération des processus inutiles ne se pose pas mais on n'exploite pas forcément tout le parallélisme intrinsèque d'un programme. Comme précédemment, les graphes sont représentés par des paquets; un processeur recevant un paquet applique le combinateur principal ce qui revient généralement à activer d'autres paquets dans un ordre dépendant de ce combinateur (parallèlement s'il est strict). Le réseau d'interconnection est un r-n.cube ([SLEEP 81b]) qui permet une diffusion exponentielle des tâches et il peut être multiplié sans augmenter le nombre de connections de chaque processeur. De plus ce réseau est homogène dans le sens où chaque processeur en a la même vue d'ensemble. Ce point est important car il permet d'éviter les "goulots d'étranglement" qui peuvent exister, dans des architectures en arbre par exemple. Le procédé de répartition des tâches sur le réseau est original: quand un processeur crée des processus, il les place dans la file des processus "suspendus". Si la tâche active ne peut se poursuivre par manque de données, elle est placée dans la file des "bloqués" et le processeur en choisit un autre parmi les "suspendus". Cependant ces dernières peuvent également être exécutées par les processeurs fils qui les subtilisent à leur père et les placent dans leur file des "actifs". Ces "actifs" devront être exécutés par ce processeur et ne pourront plus être subtilisés; ceci assure qu'un argument ne peut être éloigné du processeur demandeur mais la contrepartie est que le réseau peut être chargé de façon irrégulière. Le langage utilisé dans ce système est SASL

([TURNER 81b]).

Le système AMPS (Applicative Multiprocessing System) ([KELLER 79]) présente de nombreuses similitudes avec le précédent. Le langage utilisé est un dialecte de LISP et le schéma d'exécution est le même que ZAPP. Le réseau se présente ici sous forme d'arbre; les processeurs situés aux noeuds se chargent de la répartition des tâches et de la communication tandis que l'exécution elle-même se déroule au niveau des feuilles. Les programmes sont compilés sous forme de blocs qui sont affectés à des processeurs-feuilles. La structure de blocs exploite la notion de localité qui est contenue dans les programmes afin de réduire la communication (exécution sur un seul processeur) et l'espace adresse requis (adressage local). Elle permet de plus de mettre en oeuvre un "grain" de parallélisme équilibrant le coût de la communication introduite avec le gain dû à la simultanéité des tâches: c'est ainsi que des tâches ne sont jamais créées pour des opérations primitives car leur temps d'exécution n'est pas suffisamment élevé pour compenser le coût de la communication. Les instructions d'un bloc contiennent les adresses des arguments (relatives au bloc) et une liste de signaux ("notifiers") indiquant les adresses des instructions en attente de ce résultat. Il existe une instruction de création d'un autre bloc ("invoke") qui utilise un champ d'adresse global pour permettre de réaliser les liaisons entre blocs. A chaque processeur feuille sont associées deux listes: celles des opérations à effectuer et celle des résultats calculés avec leur destination. Le réseau de communication sert d'une part à transmettre ces résultats et d'autre part à répartir les travaux entre les listes des différentes feuilles; chaque processeur noeud reçoit pour cela périodiquement des signaux indiquant l'état d'occupation de sa descendance et il peut alors décider de transférer les tâches en attente dans la liste d'un processeur vers la liste d'un voisin moins

chargé. Notons que par rapport au schéma précédent, les tâches sont toujours uniformément réparties, cependant le surcroît de gestion est plus important ici puisqu'une grande partie des processeurs ne participe pas au calcul efficace et que des évaluations de charges sont nécessaires.

Le multiprocesseur de Grit et Page déjà mentionné plus haut ([GRIT 80b]) est une architecture faiblement couplée implémentant le schéma de "full substitution". Il s'agit d'un modèle data-flow composé d'ensembles de processeurs, de mémoires et d'allocateurs de ressources. Ce modèle a été utilisé pour simuler différentes structures en faisant varier certaines paramètres (nombre de processeurs, de mémoire, réseau d'interconnection,...) et étudier le gain obtenu en temps d'exécution en fonction du nombre de processeurs alloués. Dans le meilleur des cas cette croissance est linéaire jusqu'à épuisement du parallélisme intrinsèque du programme, le principal problème étant d'éviter les "goulots d'étranglement" et de diminuer les délais de communication.

Un premier exemple de machine à réduction est celle de Mago ([MAGO 80]) qui est destinée à exécuter le langage FP. L'architecture physique a la forme d'un arbre composé de deux types de cellules: les cellules L aux feuilles qui représentent la mémoire et les cellules T aux noeuds qui servent au calcul et à la communication. Un programme est distribué de façon linéaire à raison d'un symbole FP par cellule feuille. Le schéma d'exécution des réductions est "parallel innermost": toutes les expressions réductibles les plus imbriquées sont réduites simultanément. L'exécution en elle-même est assurée par des micro-programmes qui sont répartis sur les cellules L. Le principal problème intervient lorsque les expressions résultantes sont plus volumineuses que les formes initiales; il faut alors procéder à des décalages qui sont réalisés à

chaque cycle et par l'ensemble des cellules. Chaque cellule possède sa propre horloge et l'ensemble reste dans un état cohérent grâce aux communications qui balayent l'arbre de bas en haut et de haut en bas. Les cycles ne correspondent en fait qu'à des instantanés compréhensibles de l'utilisateur. On constate donc que si cette machine paraît moins souple que les architectures faiblement couplées citées auparavant, elle possède l'avantage d'être plus directement adaptée au langage qu'elle supporte et que le coût de gestion introduit est plus faible.

La machine à réduction de Newcastle ([TRELEAVEN 80]) implémente aussi le schéma "parallel-innermost" dans un modèle à réduction par chaîne. Elle est composée d'une mémoire globale contenant les définitions et d'une boucle constituée de l'expression à réduire située dans des registres disposés en alternance avec les unités d'exécution. Celles-ci peuvent accéder la mémoire et les deux registres qui leurs sont reliés à travers deux files. Chaque unité d'exécution possède des registres contenant des informations sur la sous-expression traversée (registre tampon, registre d'entrée) et des tables de transition indiquant les actions à effectuer au vu des symboles rencontrés (contenus dans le tampon) et du symbole lu (registre d'entrée); ces actions peuvent être: changer l'état, changer la direction de lecture, charger le buffer, le décharger vers un registre externe si l'expression est réduite... Le rôle de chaque processeur est de trouver une expression réductible, de la réduire, de la transmettre et ainsi de suite. Il peut pour cela échanger (envoyer, recevoir) des symboles avec ses deux registres voisins. Un tel schéma peut produire une étreinte fatale ou la famine de certaines processeurs; ces cas doivent être évités par un bon choix des tables de transition. La particularité de cette machine est qu'elle peut exécuter n'importe quel langage de réduction

pourvu qu'on lui fournisse les tables adéquates; de plus ces tables peuvent être engendrées automatiquement à partir de la syntaxe BNF du langage.

La machine GMD ([BERLING 71], [BERLING 75]) fonctionne selon le principe de réduction par chaîne mais à partir d'une organisation centralisée; le langage utilisé est basé sur le lambda-calcul et les programmes sont représentés par des expressions pré-fixées; si elle n'est pas un atome, une expression est composée de trois parties: constructeur, la fonction et l'argument. Selon que l'on désire réduire l'argument avant la fonction ou l'inverse ces sous-expressions sont placées dans l'ordre constructeur-argument-fonction ou constructeur-fonction-argument; c'est le constructeur qui permet de distinguer la fonction de l'argument. La machine est constituée de quatre unités de réduction, d'un ensemble de piles et d'un bus assurant la communication entre les unités. Le parcours des chaînes de symboles est effectué par l'unité TRANS; simultanément l'unité REDREC recherche une expression à réduire et provoque l'arrêt de la précédente dès qu'elle l'a trouvée; REDEX reçoit alors l'expression qu'elle réduit en faisant éventuellement appel à ARITH qui réalise les opérations arithmétiques. Les piles sont utilisées lors des parcours d'expressions et des réductions; la chaîne initiale et le résultat sont contenus dans une pile.

On peut citer encore la machine M3L ([SANSONNET 80]) également centralisée et qui est adaptée au traitement des listes. Elle dispose pour cela d'une mémoire de paires (car et cdr plus un descripteur) et d'un mécanisme spécial pour la récursivité permettant la sauvegarde d'un nombre limité de registres (4) et le retour d'une procédure à un niveau quelconque d'appel antérieur (mécanisme d'escape). Cette machine permet d'utiliser LISP avec des gains de temps très

importants (facteur 10 par rapport au LISP CII-HB sur IRIS 80).

Il convient également d'évoquer, bien que ce ne soit pas à proprem parler une proposition d'architecture, la SK machine de Turner ([TURNER 79]). s'agit en fait d'une nouvelle technique de mise en oeuvre impliquant compilation du programme fonctionnel de départ (en SASL ici) en une forme sa variables à base de combinateurs. Des résultats de logique ont montré que cet traduction est toujours possible et des algorithmes existent qui réalisent cet transformation ([BURGE 76]). Turner a même proposé un algorithme d'optimisati permettant de réduire la taille de l'expression obtenue ([TURNER 79]). modèle proposé met en oeuvre la transformation de graphes par réductio "left-most". Le programme compilé est représenté sous forme d'arbre dont le noeuds sont des applications de fonctions et les feuilles des constantes L'exécution consiste en une suite de réductions sur le sommet d'une pile qu contient initialement une référence à l'expression à calculer. Si c'est un application, la partie gauche (fonction) est empilée; s'il s'agit d'u combinateur il est appliqué, utilisant les arguments qui se trouvent au-dessus de lui dans la pile. Son exécution entraîne des modifications dans les sous-arbres référencés par ses arguments et ceux-ci ne seront pas calculés s'ils sont utilisés à nouveau: le programme s'auto-modifie en quelque sorte. Ce qui est considéré comme une nuisance dans les langages traditionnels ne l'est pas ici à cause de l'absence d'effets de bord: une expression est seulement simplifiée. C'est la demande d'impression qui déclanche en cascade les réductions des expressions nécessaires: il s'agit donc d'un schéma d'appel par nécessité. Ce procédé se révèle moins efficace que les modèles classiques (machine SECD ([LANDIN 64])) dans le cas général mais il met en oeuvre de façon naturelle un langage plus puissant (non strict) puisqu'il peut tolérer les

listes infinies et si on modifie les interpréteurs classiques ([BURGE 76], [TURNER 79]) pour implémenter un schéma d'appel par nécessité, le rapport de force est largement renversé. Ce type de représentation est connu depuis fort longtemps ([CURRY 58]) mais fut longtemps délaissé car il implique une multiplication du volume des programmes ce qui les rend incompréhensibles et coûteux en mémoire.

On note désormais un regain d'intérêt pour cette technique, inauguré par Turner, et qui est dû en partie au progrès de la technique en ce qui concerne les VLSI; les combinateurs sont en effet très adaptés à cette technologie puisqu'ils conduisent à des expressions formées d'un nombre réduit de symboles d'arité fixe et reproduits un grand nombre de fois. On voit donc ici que les recherches au niveau des langages fonctionnels et au niveau du matériel se rejoignent ce qui semble un signe d'espoir pour les uns et pour les autres; à quoi bon en effet disposer d'un langage qui fait perdre à l'exécution le temps gagné lors de la mise au point ou d'une machine très efficace si le temps perdu à la programmation n'est pas compensé par celui qui est économisé à l'exécution. La solution intermédiaire qui consiste à utiliser un langage de programmation et un langage à l'exécution (cas général actuellement) possède l'inconvénient d'introduire des niveaux intermédiaires (donc des surcoûts) parfois importants, de masquer à l'utilisateur le comportement exact de son programme à l'exécution (favorisant donc une programmation inefficace) et de compliquer la recherche des causes d'erreurs quand il s'en produit. L'idéal ne serait-il pas un langage simple à manipuler possédant les propriétés mathématiques nécessaires et exécuté directement par la machine? Les architectures que nous venons de citer montrent que, même si des progrès restent à faire, la voie semble toute tracée dans cette direction.

CONCLUSION.

PROBLEMES OUVERTS ET PERSPECTIVES.

Il est certain que des problèmes restent posés à propos des langages fonctionnels, en premier lieu au niveau de la mise en oeuvre, même s'ils peuvent désormais être aussi efficaces que les langages conventionnels. La question est en effet complexe: comment exécuter une expression fonctionnelle de façon optimale au sens du temps de réponse obtenu, du coût du matériel sous-jacent, de l'occupation des ressources...Nous avons vu qu'il est possible de décomposer le problème en trois parties: la représentation du programme, le schéma d'évaluation et la structure de machine sous-jacente. Ces trois aspects sont cependant très liés les uns aux autres et aucune proposition ne peut prétendre être idéale dans tous les cas d'applications.

La mesure même de la complexité des algorithmes est un problème crucial car, contrairement aux langages conventionnels très proches des machines Von Neumann bien connues, les langages fonctionnels offrent très peu d'indications sur l'efficacité de leurs programmes, ceci est naturel vue la prétention de ces langages d'ignorer les considérations de bas niveau, mais il paraît tout de même souhaitable d'indiquer au programmeur, sous une forme quelconque, une approximation de la complexité de son programme, relative aux choix de mise en oeuvre effectués.

Un tel outil semble indispensable tout du moins tant qu'il n'est pas possible de produire de façon automatique la version la plus efficace d'un programme à partir d'une définition quelconque. Il a déjà été évoqué le problème de la recherche des erreurs dans le cas fonctionnel; là aussi l'éloignement du programmeur par rapport à l'exécution réelle de son programme en machine (ordre d'évaluation, position des données en mémoire,...) est source de complications; il faut en effet déterminer l'état du système au moment de l'anomalie (quelles sont les expressions qui ont été évaluées...) et identifier l'expression qui

s'est mal terminée. Les cas d'erreurs détectés sont souvent représentés par valeur particulière ([) qui est absorbante et se propage donc jusqu'au résul final (FP). Une solution élégante adoptée par VAL ([MACGLAW 82]) est d'utili plusieurs symboles représentant les différents types d'erreurs possibi division par zéro, résultat trop élevé, trop faible, inconnu,... Le problème cependant que ces valeurs peuvent perdre de l'information en se propageant, exemple:

résultat trop élevé + résultat trop faible = inconnu

résultat trop élevé - 1 = inconnu

Cette faiblesse au niveau de la recherche des erreurs à l'exécution e généralement justifiée par le fait que les langages fonctionnels permettent diminuer leur nombre d'erreurs et de les identifier aisément à partir programme source. Cet argument est juste mais n'exclut pas pour autant un out de mise au point, même sommaire.

Un dernier obstacle au développement des langages fonctionnels concerne l entrées/sorties qui sont généralement très limitées; l'interaction du program et de son environnement se réduit souvent à la demande d'évaluation d'u expression et l'envoi du résultat; ces langages ne sont pas "history-sensitive comme le mentionne Backus ([BACKUS 78]) qui a proposé un langage AS introduisant la notion d'état. D'autres propositions ont été faites pou résoudre cette difficulté ([HENDERSON 81], [FRIEDMAN 80], [BURSTALL 80]) mai les outils d'entrée/sortie sont généralement introduits au détriment de propriétés qui font le charme des langages fonctionnels. Il est possibl d'introduire des fonctions telles que lireconsole, écrireconsole écrirefichier1,... mais si elles sont banalisées elles risquent d'invalider u

bon nombre de lois du langage. En effet les fonctions de lecture, si elle sont combinées comme les autres ont un comportement variable indépendant des arguments; par exemple la loi:

eq o [f;f] ≡ "T"

n'est plus vraie en FP car f peut être égale à lireconsole par exemple. De même les fonctions d'écriture produisent un effet de bord évident. Les langages fonctionnels étant intrinsèquement allergiques aux effets de bord, il semblerait plus sain de décomposer le langage en deux parties: un sous-ensemble applicatif utilisable tant qu'il n'y a pas d'entrées/sorties et un sur-ensemble servant uniquement à introduire des ordres explicites de lecture et d'écriture, les lois des langages fonctionnels restent appliquables à leur niveau il faut alors en exhiber d'autres pour le sur-langage.

Les langages fonctionnels ne sont pas la seule tentative de programmation plus proche des conventions mathématiques. LUCID par exemple ([ASHCROFT 77]) est basé sur la notion de suite. Les variables sont représentées par la suite des valeurs qu'elles représentent, les constantes par une suite constante et toutes les opérations manipulent des suites. Même si la sémantique de ces objets est fondamentalement différente de celle des langages conventionnels, les programmes LUCID ne présentent pas vis-à-vis de ceux-ci de différences exceptionnelles (boucles, pseudo-affectation transformée en calcul de suites). Les preuves de programmes sont évidemment plus faciles que dans le cadre traditionnel mais deviennent vite assez lourdes car elles sont essentiellement constituées de manipulations de suites où le raisonnement par récurrence s'impose. L'interprétation parallèle de langages basés sur la notion de suite semble également moins immédiate que dans le cadre des langages fonctionnels.

On a donc pu constater que la programmation fonctionnelle se trouve à

l'intersection de deux domaines pleins de promesses, à savoir:

- l'expression des problèmes: ils permettent des preuves et des transformations de programmes pouvant aller jusqu'à la synthèse de spécifications non exécutables ([CLARK 80], [DARLINGTON 81a]),
- la mise en oeuvre: ils constituent un moyen simple et élégant d'exprimer des algorithmes parallèles sans se soucier des difficultés généralement rencontrées dans ce domaine (synchronisation, variables partagées...).

Les seuls obstacles s'opposant jusqu'à présent à leur développement étaient leur inefficacité et la programmation inhabituelle qu'ils impliquaient. Le premier appartient désormais au passé et le second ne tient qu'au fait que les programmeurs sont trop accoutumés aux langages actuels. Avec l'apparition de techniques de mise en oeuvre très performantes et de nouvelles structures de machine permettant une exécution au moins aussi rapide que les langages conventionnels on peut espérer que les progrès apportés par la programmation fonctionnelle vont être désormais plus largement reconnus. Il suffit pour s'en convaincre de constater le développement croissant de langages fonctionnels comme HOPE et ML.

REMERCIEMENTS

Je remercie particulièrement J-P. Banâtre pour les lectures qu'il a faites de ce document et les conseils judicieux qu'ils m'a apportés. J'adresse aussi mes remerciements à L. Kott dont les remarques ont permit d'éviter quelques oublis. Une discussion avec G. Berry m'a également permit de combler quelques lacunes. Je tiens également à remercier B. Maret qui a assuré avec efficacité la dactylographie de ce document en utilisant l'éditeur de textes de Multics.

BIBLIOGRAPHIE

[ALLEN 78] Allen J., Anatomy of LISP, Mc Graw-Hill (1978).

[ASHCROFT 77] Ashcroft E.A., Wadge W.W., Lucid, a Nonprocedural Language with Iteration, Comm. ACM, Vol 20,7 (July 1977), pp. 519-526.

[ASCHCROFT 82] Aschcroft E.A., Wadge W.W., R for Semantics, ACM Trans. on Programming Languages and Systems, Vol. 4, 2 (April 1982), pp. 283-294.

[BACKUS 78] Backus J.,Can Programming be Liberated from the Von Neumann style ? A Functional Style and its Algebra of Programs, Comm. ACM, Vol. 21, 8 (August 1978), pp. 613-641.

[BACKUS 81] Backus J., The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions, Lectures Notes in Computer Science, No 107, Springer-Verlag, Heidelberg.

[BAKER 77] Baker H.G. Jr, Hewitt C., The Incremental Garbage Collection of Processes, SIGPLAN Notices, Vol 12,8 (August 1977), pp. 55-59.

[BAKER 78] Baker H.G. Jr, List Processing in Real Time on a Serial Computer, Comm. ACM, Vol. 21,4 (April 1978), pp. 280-294.

[BERKLING 71] Berkling K.J., A Computing Machine based on Tree Structures, IEEE Trans. on Computers, Vol C-20, 4 (April 1971).

[BERKLING 75] Berkling K., Reduction Languages for Reduction Machines, Proc. 2nd. Int. Symp. Computer Architecture (1975), pp. 133-140.

[BERRY 79] Berry G., Levy J-J., Minimal and Optimal Computations of Recursive Programs, J. of the ACM, Vol. 26,1 (January 1979), pp. 148-175.

[BERRY 81] Berry G., Programming with Concrete Data Structures and Sequential

REMERCIEMENTS

Je remercie particulièrement J-P. Banâtre pour les lectures qu'il a faites de ce document et les conseils judicieux qu'ils m'a apportés. J'adresse aussi mes remerciements à L. Kott dont les remarques ont permit d'éviter quelques oublis. Une discussion avec O. Berry m'a également permit de combler quelques lacunes. Je tiens également à remercier B. Maret qui a assuré avec efficacité la dactylographie de ce document en utilisant l'éditeur de textes de Multics.

BIBLIOGRAPHIE

[ALLEN 78] Allen J., Anatomy of LISP, Mc Graw-Hill (1978).

[ASHCROFT 77] Ashcroft E.A., Wadge W.W., Lucid, a Nonprocedural Language with Iteration, Comm. ACM, Vol 20,7 (July 1977), pp. 519-526.

[ASCHCROFT 82] Ashcroft E.A., Wadge W.W., R for Semantics, ACM Trans. on Programming Languages and Systems, Vol. 4, 2 (April 1982), pp. 283-294.

[BACKUS 78] Backus J.,Can Programming be Liberated from the Von Neumann style ? A Functional Style and its Algebra of Programs, Comm. ACM, Vol. 21, 8 (August 1978), pp. 613-641.

[BACKUS 81] Backus J., The Algebra of Functional Programs: Function Level Reasoning, Linear Equations, and Extended Definitions, Lectures Notes in Computer Science, No 107, Springer-Verlag, Heidelberg.

[BAKER 77] Baker H.G. Jr, Hewitt C., The Incremental Garbage Collection of Processes, SIGPLAN Notices, Vol 12,8 (August 1977), pp. 55-59.

[BAKER 78] Baker H.G. Jr, List Processing in Real Time on a Serial Computer, Comm. ACM, Vol. 21,4 (April 1978), pp. 280-294.

[BERKLING 71] Berkling K.J., A Computing Machine based on Tree Structures, IEEE Trans. on Computers, Vol C-20, 4 (April 1971).

[BERKLING 75] Berkling K., Reduction Languages for Reduction Machines, Proc. 2nd. Int. Symp. Computer Architecture (1975), pp. 133-140.

[BERRY 79] Berry G., Levy J-J., Minimal and Optimal Computations of Recursive Programs, J. of the ACM, Vol. 26,1 (January 1979), pp. 148-175.

[BERRY 81] Berry G., Programming with Concrete Data Structures and Sequential

Algorithms, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture (October 1981), pp. 49-57.

[BOYER 75] Boyer R., Moore J., Proving Theorems about LISP Function, J. of ACM, Vol 22,1 (January 1975), pp. 129-144.

[BURGE 76] Burge W., Recursive Programming Techniques, Addison Wesley, Reading, Mass (1976).

[BURSTALL 69] Burstall R., Proving Properties of Programs by Structural Induction, Computer J., Vol 12,1 (February 1969), pp 41-48.

[BURSTALL 77] Burstall R., Darlington J., A Transformation System for developing Recursive Programs, Journal of the ACM, Vol. 24, 1 (January 1977), pp. 44-67.

[BURSTALL 80] Burstall R.M., MacQueen D.B., Sannella D.T., HOPE: an Experimental Applicative Language, Internal Report, CSR-62-80 (May 1980), University of Edinburgh.

[CADIOU 72] Cadiou J.M., Recursive Definitions of Partial Functions and their Computations, Ph.D.Th, Standford University (1972).

[CHIARINI 80] Chiarini A., On FP Languages Combining Forms, SIGPLAN Notices, Vol 15,9 (September 1980), pp. 25-27.

[CHURCH 41] Church A., The Calculi of Lambda Conversion, Ann. of Math. Studies, Vol. 6, N.J. Princeton University Press (1941).

[CLARK 77] Clark D.W., Green C.C., An Empirical of List Structure in LISP, Comm. ACM, Vol 20,2 (February 1977), pp. 78-87.

[CLARK 80] Clark K.L., Darlington J., Algorithm Classification through Synthesis, Computer J., Vol 23,1 (1980), pp. 61-65.

[COLLINS 60] Collins G.E., A Method for Overlapping and Erasure of Lists, Comm. ACM, Vol 3,12 (December 1960), pp. 655-657.

[CURRY 58] Curry H.B., Feys R., Combinatory Logic, Vol. I, North Holland (1958).

[DARLINGTON 76] Darlington J., Burstall R.M., A system which automatically improves Programs, Acta Informatica, Vol 6,1 (1976), pp. 41-60.

[DARLINGTON 81a] Darlington J., Program Transformation, Course on Functional Programming and its Applications (July 1981), University of Newcastle upon Tyne.

[DARLINGTON 81b] Darlington J., Reeve M., ALICE a Multiprocessor Reduction Machine for the Parallel Evaluation of Applicatives Languages, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture (October 81), pp.65-75.

[DENNIS 75] Dennis J. B., Packet Communication Architecture, Proc. 1975 Comp. Conf. on Parallel Processing (1975), pp. 224-229.

[DIJKSTRA 75] Dijkstra E.W., Lamport L., Martin A.J., Schlotan C.S., Steffens E.F.M., On-the-fly garbage collection: an exercice in cooperation, note EWD496 (June 1975).

[DOWNEY 76] Downey P.J., Sethi R., Correct Computation Rules for Recursive Languages, SIAM J. Comput., Vol. 5,3 (September 1976), pp. 378-401.

[FEATHER 82] Feather M.S., A System for Assisting Program Transformation, ACM Trans. on Programming Languages and Systems, Vol 4,1 (January 1982), pp. 1-20.

[FLOYD 67] Floyd R.W., Assigning Meanings to Programs, Proc. Symp. in Applied Mathematics, Vol. 19, Mathematical Aspects of Computer Science, American Mathematical Society (1967).

[FRIEDMAN 76] Friedman D.P., Wise D.S., CONS should not evaluate its Arguments, Int. Conf. on Automata, Languages and Programming, Edinburgh University Press (1976), pp. 257-284.

[FRIEDMAN 78] Friedman D.P., Wise D.S., Aspects of Applicative Programming for Parallel Processing, IEEE trans. on Computers, Vol C-27, 4 (April 1978), pp. 289-295.

[FRIEDMAN 80] Friedman D.P., Wise D.S., An indeterminate Constructor for Applicative Programming, 7th ACM Symp. on Principles of Programming Languages (1980), pp. 245-250.

[GORDON 79] Gordon M., Milner R., Wadsworth C., Edinburgh LCF, LNCS No 78, Springer-Verlag (1979).

[GRIT 80a] Grit D., Page R., Performance of a multiprocessor for Applicative Programs, Proc. Performanc (May 1980), pp. 181-189.

[GRIT 80b] Grit D., Page R., A multiprocessor Model for parallel Evalmation of applicative Programs, J. of Digital Systems, Vol 4, 2 (Summer 1980), pp. 135-151.

[GRIT 81] Grit D., Page R., Deleting Irrelevant Tasks in an Expression-Oriented Multiprocessor System, ACM Trans. on Programming Languages and Systems, Vol. 3, 1 (January 1981), pp. 48-59.

[GUTTAG 81] Guttag J.V., Horning J., Williams J., FP with Data Abstraction and Strong Typing, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture (October 1981), pp. 11-24.

[HABERMANN 80] Habermann A.N., Notes on Programatics and its language Alfa, Carnegie-Mellon University Pittsburgh.

[HENDERSON 76] Henderson P., Morris J.H., A lazy Evaluator, third ACM Symp. on Principles of Programming Languages (1976), pp. 95-103.

[HENDERSON 81] Henderson P., Purely Functional Operating Systems, Advanced Course on Functional Programming and its Applications, University of Newcastle upon Tyne (July 1981).

[HOARE 69] Hoare C.A.R, An Axiomatic Basis for Computer Programming, Comm. ACM, Vol. 12, 10 (October 1969), pp.576-583.

[HUET 79] Huet G., Levy J-J., Call By need Computations in non-ambiguous linear Term Rewriting Systems, rapport de Recherche Laboria No 359 (Août 1979), IRIA.

[HUET 80] Huet G., Hullot J-M., Proofs by induction in Equational Theories with Constructors, 21st. Symp. on Fondations of Computer Science(1980), pp. 96-107.

[IVERSON 62] Iverson K.E., A Programming Language, Wiley, New-York (1962).

[IVERSON 79] Iverson K.E., Operators, ACM Trans. on Programming Languages and Systems, Vol 1,2 (October 1979), pp. 161-176.

[KELLER 79] Keller R. M., Lindstrom G., Patil S., A Loosely-coupled applicative multiprocessing system, Proc. 1979 AFIPS NCC, Vol. 48 (1979), pp. 613-622.

[KNUTH 75] Knuth D.E., The Art of Computer Programming I, Fundamental Algorithms, Addison-Wesley, Reading, MA, Section 2.3.5 (1975).

[KOTT 80] Kott L., Des substitutions dans les systemes d'équations algebriques sur le magma. Application aux transformations de programmes et a leur correction, Thèse, Université de Paris 7 (1980).

[LANDIN 64] Landin P.J., The Mechanical Evaluation of Expressions, Computer J., Vol. 6,4 (April 1964), pp. 308-320.

[LANDIN 66] Landin P.J., The next 700 Programming Languages, Comm. ACM, Vol. 9,3 (March 1966),pp. 157-166.

[LEROY 74] Leroy H., La Fiabilité des Programmes, Travaux de l'Institut d'Informatique de Namur, No 3 (1974).

[LESZCZYLOWSKI 80] Leszczylowski J., Theory of FP Systems in Edinburgh LCF, Internal Report, CSR 61-80 (April 1980), University of Edinburgh.

[LEVY 78] Levy J-J., Réductions correctes et optimales dans le lambda-calcul, Thèse, Université de Paris 7 (1978).

[MACCARTHY 60] Mac Carthy J., Recursive Functions of Symbolic Expressions and Their Computation by Machine Part 1, Comm. ACM, Vol. 3,4 (April 1960), pp. 184-195.

[MACCARTHY 63] Mac Carthy J., A Basis for a Mathematical Theory of Computation, Computer Programming and Formal Systems, (1963).

[MACGRAW82] Mac Graw J.R., The VAL Language: Description and Analysis, ACM Trans. on Programming Languages and Systems, Vol 4,1 (January 1982), pp. 44-82.

[MAGO 80] Mago G.A., A cellular Computer Architecture for functional Programming, Proc. IEEE COMPCON 80 (February 1980).

[MANNA 73] Manna Z., Ness S., Vuillemin J., Inductive Methods for Proving Properties of Programs, Comm. ACM, Vol 16,8 (August 1973), pp. 491-502.

[MANNA 75] Manna Z., Waldinger R., Knowledge and reasoning in Program Synthesis, Artif. Intell. J., Vol. 6,2 (1975), pp. 175-208.

[MANNA 77] Manna Z., A New Approach to Recursive Programs, Perspectives on Computer Science, Anita K. Jones, Academic Press, INC (1977), pp. 103-124.

[MORRIS 80] Morris J.H., Schmidt E., Wadler P., Experience with an Applicative String Processing Language, Proc. 7th ACM Symposium on Principles of Programming Languages (1980), pp. 32-46.

[MYCROFT 80] Mycroft A., Theory and Practice of Transforming Call-by-need into Call-by-value, 4th Int. Symp. on Programming (1980).

[PETTOROSSI 81] Pettorossi A., A Transformational Approach for Developing Parallel Programs, Proc. COMPAR 81 (June 1981) et Lecture Notes in Computer Science No 111, pp. 245-258.

[PRINI 80] Prini G., Explicit Parallelism in LISP-like Languages, Conf. Record of the 1980 LISP Conf., Standford (August 1980), pp. 13-18.

[RAYMOND 75] Raymond F.H., Note sur l'algèbre des fonctions, R.A.I.R.O., Vol R-3 (December 1975).

[REYNOLDS 70] Reynolds J.C., A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept, Comm. ACM, Vol 13,5 (May 1970),pp. 308-319.

[ROUSSEL 76] Roussel P., PROLOG, Manuel de référence et d'utilisation (Septembre 1976), GIA Marseille-Luminy.

[SANSONNET 80] Sansonnet J.P., Castan M., Percebois C., M3L: A List-directed Architecture, Proc. Symp. on Computer Architecture (May 1980), pp. 105-112.

[SCHONFINKEL 24] Schonfinkel M., Über die Bausteine der Mathematischen Logik, Math. Annalen, Vol 92,305 (1924).

[SLEEP 81a] Sleep M.R., Burton F.W., Towards a Zero Assignment Parallel Processor, Proc. 2nd Int. Conf. on Distributed Computing (April 1981), pp. 80-85.

[SLEEP 81b] Sleep M.R., Burton F.W., Executing Functional Programs on a Virtual Tree of Processors, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture (October 1981), pp. 187-194.

[STOY 81] Stoy J.E., Some Mathematical Aspects of Functional Programming, Advanced Course on Functional Programming and its Applications, University of Newcastle upon Tyne (July 1981).

[SUSSMAN 81] Sussman G.J., Notes on LISP, Advanced Course on Functional Programming and its Applications, University of Newcastle upon Tyne (July 1981).

[TRELEAVEN 80] Treleaven P.C., A Multiprocessor Reduction Machine for

User-defined Reduction Languages, Proc. Seventh Int. Symp. on Computer Architecture (May 1980) ,pp. 121-130.

[TRELEAVEN 82] Treleaven P.C., Brownbridge D.R., Hopkins R.P. , Data Driven and Demand Driven Computer Architecture, ACM Computing Surveys, Vol. 14, 1 (March 1982).

[TURNER 79] Turner D.A., A New Implementation Technique for Applicative Languages, Software-Practice and Experience, Vol 9 (1979), pp. 31-49.

[TURNER 81a] Turner D.A., The Semantic Elegance of Applicative Languages, Proc. of the 1981 Conf. on Functional Programming Languages and Computer Architecture (October 1981), pp. 85-92.

[TURNER 81b] Turner D.A., Abramson H., InterSASL Reference Manual, Technical Manual, TM-26 (September 1981), University of British Columbia Vancouver.

[VUILLEMIN 74] Vuillemin J., Correct and Optimal Implementations of Recursion in a Simple Programming Language, Journal of Computer and Systems Sciences, Vol 9,3 (December 1974), pp. 332-354.

[WADLER 81] Wadler P., Applicative Style Programming,Program Transformation, and List Operators, Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture (October 1981), pp. 25-32.

[WILLIAMS 81] Williams J.H., Notes on the FP style of Functional Programming, Advanced Course on Functional Programming Languages and Computer Architecture (October 1981), pp. 187-194.

[WILLIAMS 82] Williams J.H., On the Development of the Algebra of Functional Programs, ACM Trans. on Programming Languages and Systems, Vol 4,4 (October 1982), pp. 733- 757.

[WISE 81] Wise D.S., Interpreters for Functional Programming, Advanced Course on Functional Programming and its Applications, University of Newcastle upon

Tyne (July 1981).

PI 180 **Traitements de textes et manipulations de documents : bibliographie analytique**
J. André . 20 pages ; Septembre 1982

PI 181 **Algorithme assurant l'insertion dynamique d'un processeur autour d'un réseau à diffusion et garantissant la cohérence d'un système de numérotation des paquets global et réparti**
Annick Le Coz, Hervé Le Goff, Michel Ollivier . 31 pages ; Octobre 1982

PI 182 **Interprétation non linéaire d'un coefficient d'association entre modalités d'une juxtaposition de tables de contingence**
Israël César Lerman . 34 pages ; Novembre 1982

PI 183 **L'IRISA vu à travers les stages effectués par ses étudiants de DEA (1è année de thèse)**
Daniel Herman . 41 pages ; Novembre 1982

PI 184 **Commande non linéaire robuste des robots manipulateurs**
Claude Samson . 52 pages ; Janvier 1983

PI 185 **Dialogue et représentation des informations dans un système de messagerie intelligent**
Philippe Besnard, René Quiniou, Patrice Quinton, Patrick Saint-Dizier, Jacques Siroux, Laurent Trilling . 45 pages ; Janvier 1983

PI 186 **Analyse classificatoire d'une correspondance multiple ; typologie et régression**
I.C. Lerman . 54 pages ; Janvier 1983

PI 187 **Estimation de mouvement dans une sequence d'images de télévision en vue d'un codage avec compensation de mouvement**
Claude Labit . 132 pages ; Janvier 1983

PI 188 **Conception et réalisation d'un logiciel de saisie et restitution de cartes élémentaires**
Eric Sécher . 45 pages ; Janvier 1983

PI 189 **Etude comparative d'algorithmes pour l'amélioration de dessins au trait sur surfaces point par point**
M.A. ROY . 96 pages ; Janvier 1983

PI 190 **Généralisation de l'analyse des correspondances à la comparaison de tableaux de fréquence**
Brigitte Escofier . 35 pages ; Mars 1983

PI 191 **Association entre variables qualitatives ordinales «nettes» ou «floues»**
Israël-César Lerman . 42 pages ; Mars 1983

PI 192 **Un processeur intégré pour la reconnaissance de la parole**
Patrice Frison . 80 pages ; Mars 1983

PI 193 **The Systematic Design of Systolic Arrays**
Patrice Quinton . 39 pages ; Avril 1983

PI 194 **Régime stationnaire pour une file M/H/1 avec impatience**
Raymond Marie et Jean Pellaumail . 8 pages ; Mars 1983

PI 195 **SIGNAL : un langage pour le traitement du signal**
Paul Le Guernic, Albert Benveniste, Thierry Gautier . 49 pages ; Mars 1983

PI 196 **Algorithmes systoliques : de la théorie à la pratique**
Françoise André, Patrice Frison, Patrice Quinton . 19 pages ; Mars 1983

PI 197 **HAVANE : un système de mise en relation automatique de petites annonces**
Patrick Bosc, Michèle Courant, Sophie Robin, Laurent Trilling . 79 pages ; Mai 1983

PI 198 **Une procédure de décision en logique non-monotone**
Philippe Besnard . 59 pages ; Mai 1983

PI 199 **A formal proof system for infinitary rational expressions**

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique